

AD-A166 353

JOINT PROGRAM ON RAPID PROTOTYPING RAPIER (RAPID
PROTOTYPING TO INVESTIGA. (U) HONEYWELL INC GOLDEN
VALLEY MN COMPUTER SCIENCES CENTER 28 MAR 85

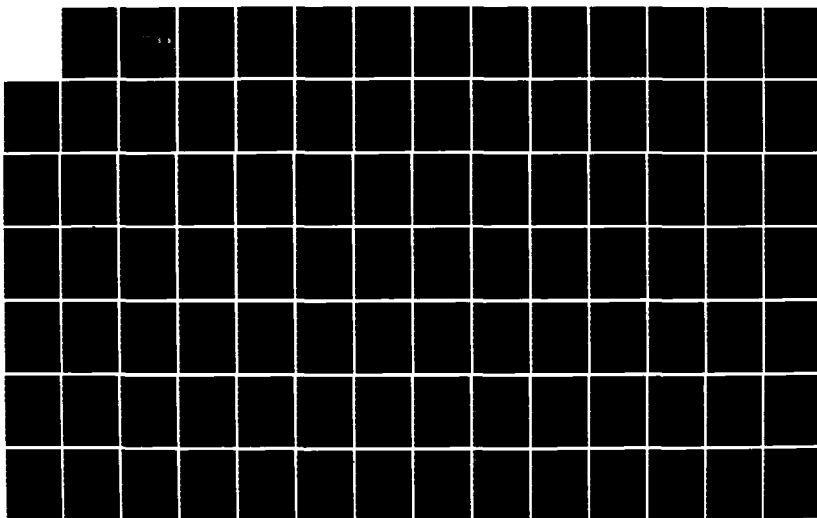
1/4

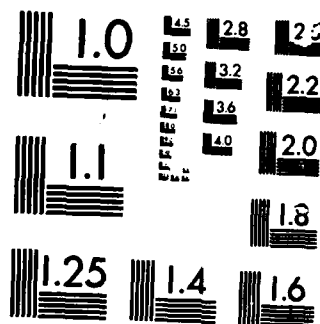
UNCLASSIFIED

N00014-85-C-0666

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART

12

AD-A166 353

Final Scientific Report to the
Office of Naval Research

Joint Program on Rapid Prototyping

RaPIER (Rapid Prototyping to Investigate End-user Requirements)

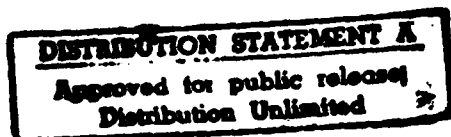
Contract No. N00014-85-C-0666
28 March 1986

RaPIER



DTIC
ELECTE
APR 07 1986
S D

DTIC FILE COPY



Honeywell
Computer Sciences Center
1000 Boone Ave. North
Golden Valley, Minnesota 55427

86 3 27 009

Final Scientific Report to the
Office of Naval Research

Joint Program on Rapid Prototyping
RaPIER (Rapid Prototyping to Investigate End-user Requirements)

Contract No. N00014-85-C-0666
28 March 1986

Honeywell Computer Sciences Center
1000 Boone Avenue North
Golden Valley, Minnesota 55427

EXECUTIVE SUMMARY

Traditional requirements definition methods consistently fail to produce requirements from which satisfactory systems can be designed and built. The RaPIER project believes that rapidly built prototypes which model critical system requirements can lead to early consensus on requirements that are acceptable to customers and feasible to implement. The project's goals are

- o to define a methodology for prototype construction and for using prototypes to investigate end-user requirements;
- o to develop a prototype of a software engineering environment that supports end-user requirements prototyping.

The environment will be based on the methodology and contain tools that support, encourage and/or enforce it. The RaPIER approach is to build prototypes from reusable Ada software parts stored in a software database, and to express them in a very high level language that specifies how the parts are tailored and interconnected to form a complete prototype.

This report presents the results of a one year ONR and internally funded investigation done to lay the technical and methodological foundations for developing and transferring the prototype of the RaPIER prototype engineering environment. The report covers work in:

- o Prototyping Methodology. Constraints which prototypes and the prototyping process must meet, a prototyping life cycle, an object-oriented prototype construction technique and a model of computation for object-oriented prototypes.
- o Software Database Management Systems. Requirements for a software base to manage reusable Ada code and companion material such as documentation and a scheme for classifying reusable parts based on logical behavior rather than physical code.
- o Ada Source Code Reusability. Characteristics which reusable Ada code must display and guidelines for achieving those characteristics.
- o Fragment Generation. An assessment of the applicability of current program transformation technology to RaPIER.
- o Research/Demonstration Examples. Descriptions of two example prototypes built using RaPIER methodology and lessons learned from these exercises.
- o RaPIER System Requirements. Initial requirements for the RaPIER prototype engineering environment and a critique of the requirements based on lessons learned from the research example prototypes.

The RaPIER project team has had much help and encouragement. Our coworkers at Honeywell's Computer Sciences Center are always willing to listen to our ideas and criticize them in depth. Our manager understands the exploratory nature

of research and the necessity of changing course to follow up promising technical results. Honeywell divisional engineers keep us constantly aware of their "real world" needs and how the RaPIER environment should meet them. Honeywell divisions provided our research examples. International Software Systems Inc. has contributed to our understanding of the capabilities of software base management systems and very high level languages for developing software from reusable parts. The text processing tools used to produce this report are due in large part to George Jelatis of the Computer Sciences Center. The RaPIER team is particularly grateful for the constant encouragement of Ms. Elizabeth (Bets) Wald of the Naval Research Laboratory, Technical Director of the STARS Application Area. The final responsibility for the quality of this work lies with the RaPIER team: Elaine N. Frankowski (Project Leader), Curtis L. Abraham, James Grindeland, Lai King Mau, Emmanuel Onuegbe, Richard St. Dennis, and Paul Stachour.

CONTENTS

	Page
1 Introduction	1
1.1 The Problem	1
1.2 The Goals	2
1.3 The Approach	3
1.4 First Year Goals	5
1.5 First Year Outcomes	6
1.6 Future Work	7
2 Background	9
2.1 Embedded Computer Systems	9
2.2 The User	10
2.3 What is a RaPIER Prototype	10
2.4 Related Work	12
2.4.1 Simulation	12
2.4.2 Application Generators	13
2.4.3 Requirements Specification Systems	14
2.4.4 Language Based Prototyping Approaches	15
2.4.5 Incremental Development	17
3 Prototyping Methodology	19
3.1 Problem Statement	19
3.2 Outcome	19
3.3 Constraints	20
3.4 A Prototyping Life Cycle	21
3.5 The Prototype Model	22
3.6 A Prototype Construction Methodology	24
3.6.1 The Operational Approach	24
3.6.1.1 Explanation and Justification	24
3.6.1.2 Implementation	26
3.6.2 The Object-Oriented Approach	26
3.6.2.1 Explanation and Justification	26
3.6.2.2 Implementation	28
3.6.3 The Construction Procedure	29
3.7 Experience	30
3.8 Future Work	31
3.9 End Notes	33
4 A Model of Computation for Prototyping	37
4.1 Problem Statement	37
4.2 Outcome	37
4.3 Three Classical Models of Computation	37
4.3.1 Batch Processing	38
4.3.2 Time Sharing	38



A-1

CONTENTS (cont)

	Page
4.3.3 Transaction Processing	39
4.4 The Conceptual Model	40
4.4.1 Requirements for the Prototyping Model	40
4.4.2 The Model	41
4.4.2.1 Objects	42
4.4.2.2 Glue	43
4.5 Implementing the Model	44
4.5.1 Objects	44
4.5.2 Glue	45
4.6 Future Work	45
5 Software Base - Classification Scheme	47
5.1 Problem Statement	47
5.2 Outcome	48
5.3 Introduction	48
5.4 The Behavior Abstraction Classification Scheme	50
5.4.1 Semantic Modelling and Object-oriented Classification	50
5.4.2 Building Prototypes With Reusable Objects	52
5.4.3 The Classification Scheme in RaPIER	53
5.5 The Phased Implementation of Behavior Abstraction	54
5.5.1 Phase 1 - Manual Classification	54
5.5.1.1 The SBMS Schema	55
5.5.1.2 Browsing	58
5.5.1.3 Classification	60
5.5.2 Phase 2 - Semi-Automatic Updates	60
5.6 Phase 3 - Behavior Abstraction	61
5.7 Interfaces	61
5.7.1 Introduction	61
5.7.2 The Browser	62
5.7.2.1 Features Needed for Browsing	62
5.7.3 The Programmatic Interface	63
5.8 Future Work	64
5.9 Appendix	65
6 Reusability Guidelines for Ada	73
6.1 Problem Statement	73
6.2 Outcome	74
6.3 Reusability Guidebook	74
6.4 Future Work	205
7 Fragment Generation Study	207
7.1 Problem Statement	207
7.2 Outcome	208
7.3 Program Transformation Systems	208
7.4 Discussion	214
7.5 Future Work	215

CONTENTS (cont)

	Page
8 Demonstration/Research Examples	217
8.1 Problem Statement	217
8.2 Outcome	217
8.3 Criteria for RaPIER Demonstration/Research Examples	218
8.3.1 The ASCLSS Example Meets the Criteria for a Demonstration Example	220
8.3.2 The IDA Example Meets the Criteria for a Demonstration Example	221
8.4 Description of the ASCLSS Demonstration Example	222
8.4.1 ASCLSS Overview	222
8.4.2 Question Under Investigation	223
8.4.3 The Prototyping Process for ASCLSS	225
8.4.3.1 Identifying Requirements to Prototype	225
8.4.3.2 Constructing The Prototype	225
8.4.3.3 Exercising the Prototype	227
8.4.3.4 Incorporating Results from Prototyping	230
8.5 Description of the IDA Prototype	230
8.5.1 IDA Overview	230
8.5.2 Question under Investigation	232
8.5.3 The Prototyping Process for IDA	232
8.5.3.1 Identifying Requirements to Prototype	233
8.5.3.2 Constructing the Prototype	233
8.5.3.3 Exercising the Prototype	234
8.5.3.4 Incorporating Results from Prototyping	235
8.6 Benefits of Examples to the RaPIER Project	235
8.6.1 Prototyping Lifecycle Benefits	235
8.6.2 RaPIER Prototype Engineering Environment Benefits	236
8.6.3 Software Base and Reusability Benefits	236
8.7 Future Work	237
 9 RaPIER Prototype Engineering Environment	 239
9.1 Problem Statement	239
9.2 Outcome	239
9.3 RaPIER Initial Requirements	240
9.3.1 Introduction	241
9.3.2 Decisions Already Made	242
9.3.3 General Principles and RaPIER-wide Requirements	243
9.3.4 RaPIER System Objectives	245
9.3.5 Contexts	246
9.3.5.1 RaPIER Top Level	246
9.3.5.2 Context Opener	247
9.3.5.3 Prototype Construction	249
9.3.5.4 Prototype Execution	251
9.3.5.5 Software Base (SWB)	252
9.3.5.6 User Help and Training	253
9.3.5.7 (!) Incorporating Results	254

CONTENTS (cont)

	Page
9.3.5.8 Services	254
9.3.6 Support Functions	254
9.3.6.1 Construction Database	255
9.3.6.2 Execution Database	256
9.3.6.3 Save Named Context	257
9.3.6.4 Kill Named Context	258
9.3.6.5 Restore Named Context	258
9.3.6.6 (!) Project Metrics	260
9.3.6.7 (!) RaPIER System Metrics	261
9.3.7 Non-automated Methods	262
9.3.8 Open Questions	262
9.4 Demonstration Evaluation	263
9.4.1 Introduction	263
9.4.2 Demonstration Evaluation	265
9.4.2.1 What tasks are needed for prototyping?	266
9.4.2.2 What functions are needed to support each task?	266
9.4.2.3 How are functions partitioned onto windows?	267
9.4.2.4 What windows are visible simultaneously?	267
9.4.2.5 Does each window support its task?	268
9.4.2.6 Miscellaneous	268
9.4.3 Comparison to Initial Requirements	268
9.4.3.1 Approach to Building the Next RaPIER Prototype	269
9.5 Future Work	270
9.6 End Notes	270
9.6.1 Objectives for the RaPIER Project and the RaPIER System	270
9.6.2 Non-Objectives for the RaPIER Project and System	271
9.6.3 The Typical RaPIER User	272
9.6.4 The RaPIER Setting	273
9.6.5 Workstations	274
9.6.6 Prototyping as Exploratory Programming	274
10 Financial Summary and Related Efforts	277
10.1 Financial Summary	277
10.2 Related Honeywell Efforts	277
10.2.1 Software Development Environments: SDE 1.0	277
10.2.2 Prototype Reusable Software Repository (RSR)	279
10.2.3 User Interface Prototyper	280
Bibliography	281

SECTION 1

INTRODUCTION

This report presents the results of work performed between July 1, 1985 and January 31, 1986 with Office of Naval Research funding under Contract No. N00014-85-C-0666, and of parallel, internally funded work performed by the Honeywell Computer Sciences Center between January 2, 1985 and January 31, 1986. These results, and results obtained in the next several years, will be applied by the RaPIER (Rapid Prototyping to Investigate End-user Requirements) project in developing a software engineering environment⁽¹⁾ to support prototyping for investigating end-user requirements. The environment supports a prototyping methodology, by which we mean ⁽¹⁾a collection of techniques, ⁽²⁾a prescribed order for applying the techniques, and ⁽³⁾reasons for the techniques and their order of application. The RaPIER methodology will eventually contain techniques for each phase in the prototyping life cycle and for the transitions between phases. The RaPIER environment will contain software tools that support, encourage, and/or enforce these procedures and techniques.

1.1 THE PROBLEM

Requirements for computer systems and software are now established early in the development life cycle with documents and reviews. A product that meets those requirements is then developed. The three major problems with this paradigm are:

- o changes to vague requirements occur later in the development cycle, necessitating expensive redevelopment;
- o missing or incorrect requirements are uncovered after requirements have been accepted, necessitating expensive redesign or reprogramming;
- o users are often subjectively dissatisfied with the final product, therefore that product does not entirely meet the need it was intended to meet.

(1) A software engineering environment is a collection of tools that is integrated in two respects: it presents users with a uniform access paradigm to all tools and allows the tools to exchange information without requiring users to take explicit action to translate that information. The RaPIER environment is also integrated with respect to a methodology; all of its tools support a single methodology.

A vehicle is needed to {1} eliminate vagueness in requirements, {2} uncover missing or incorrect requirements, and {3} get at users' subjective needs, by providing some model of initial requirements that can be criticized during the requirements definition phase of development.

The requirements investigation problem is especially acute in the DoD contractor community because of the high cost of mission critical software and because much software is developed for state-of-the-art systems with which neither developers nor customers have prior experience to guide requirements development [STARS83].

1.2 THE GOALS

We believe that a prototype(1) of (parts of) a system under development is an excellent vehicle for achieving an early consensus on requirements. Therefore, one of RaPIER's goals is to develop a prototype engineering environment. The environment will provide tools and techniques for developing modifiable prototypes quickly and inexpensively, and other tools and techniques for using prototypes in a systematic way. With such an environment, developers can build prototypes which make requirements visible and subject to investigation early in the product development life cycle. Domain, or application area, experts can then conduct experiments with the prototype in order to investigate requirements.

A second and equally important goal is to ensure transfer of the RaPIER environment from sheltered research surroundings to the production milieu. It is generally acknowledged that the software engineering community has major technology transfer problems [IEEE83]. They range from the lack of a well-defined engineering process to absorb new technology [MANLEY83], to a lack of understanding of "the human element in the software engineering equation" [PETERS83]. An important step in reducing the technology transfer problem for the RaPIER environment is to acknowledge the problem "up front," so that the solutions we devise or adopt are an organic part of the project rather than band-aids applied after the environment and methodology have been developed.

These goals are based on two assumptions: {1} prototyping will achieve faster consensus on requirements than is achieved using traditional means such as requirements reviews, and {2} systems built according to requirements reached through prototyping will meet users' needs better than systems developed from

(1) There are two views of prototypes for requirements investigation: that they are {1} working models of functional behavior used for communication between developers and domain experts, or {2} breadboard systems used for design analysis that leads to understanding performance and resource requirements. The RaPIER prototype engineering environment will be used to develop the former.

requirements obtained by traditional means. The assumptions are supported by experience in other engineering disciplines such as electrical engineering, and by [SEN82, SHEIL83].

1.3 THE APPROACH

Prototypes must be built quickly to be available during requirements analysis. They must be relatively inexpensive to build because a development program will not bear a high initial cost for requirements identification [GOMAA81] no matter how large the promised cost savings during system development and maintenance. They must be used efficiently and systematically, to ensure that the requirements questions under study are answered at reasonable cost and in a timely manner. Prototypes must also be modifiable, since prototype use will inevitably lead to changes until users agree that the prototype represents the "right" requirements.

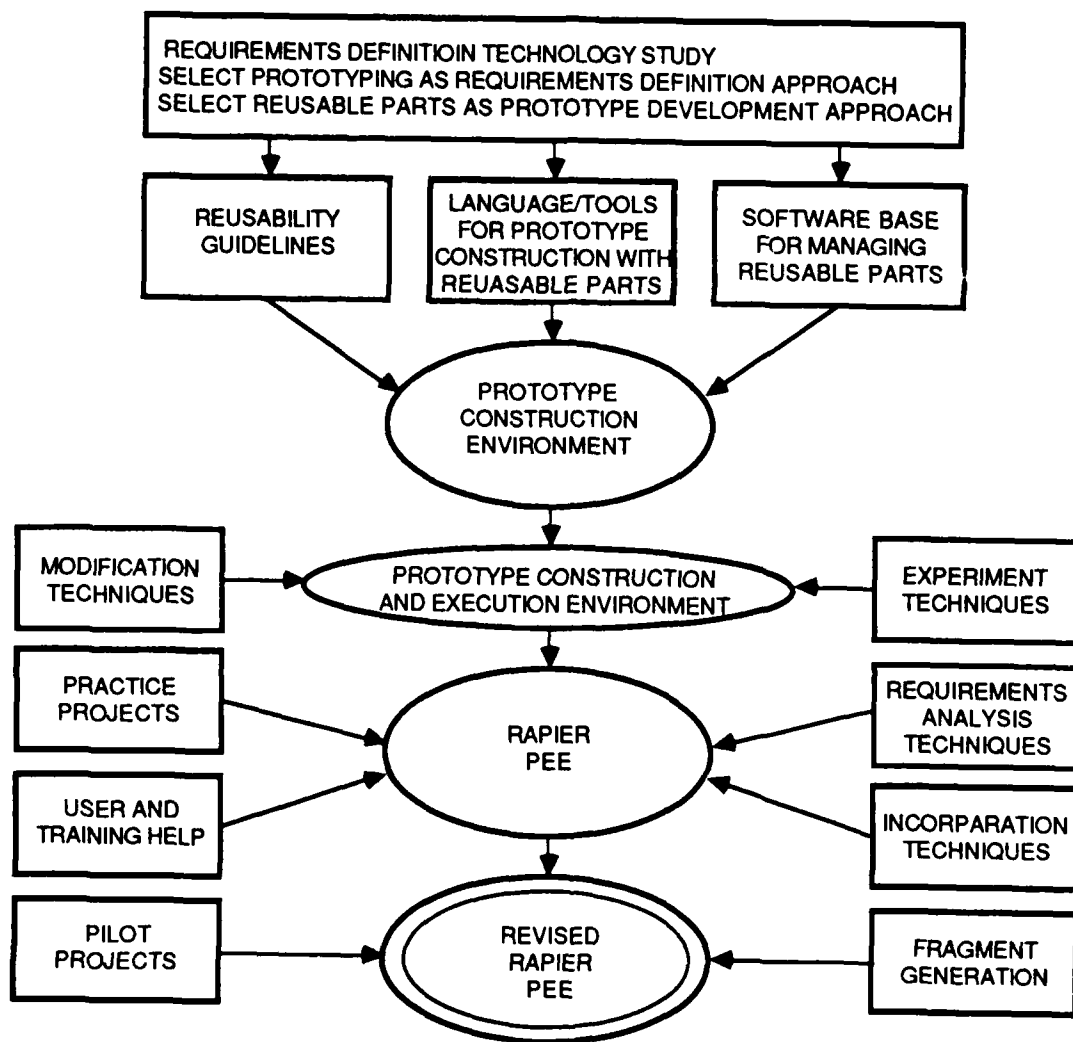
To achieve speed, low cost, and modifiability, we propose that prototypes be built from reusable software parts (source code). The reusable parts will most likely be written in Ada. Prototypes will be expressed in a very high level Prototype System Description Language (PSDL), which will eventually have a graphical syntax. Design of the prototype will be carried out in this language. PSDL identifiers will name reusable software parts and PSDL operators will specify the architecture and control structure of a prototype composed of those parts. A running prototype will be (semi-) automatically synthesized from the PSDL description.

A possible implementation of computer aided prototype synthesis from reusable parts is that the parts are stored in a software database [ONUEGBE85a, FRANKOWSKI85], that PSDL operators denote code templates that "glue" reusable parts into a running program, and that a PSDL processor "compiles" a design into a running prototype by incorporating the reusable parts into the code templates. (See [FRANKOWSKI85] for a more complete explanation of this process.)

Efficient and systematic prototype use can be achieved by including techniques for designing prototyping experiments in the requirements analysis methodology, and defining a prototype use methodology which ensures that experiments are conducted systematically.

To achieve the technology transfer goal, and obviously to educate ourselves about prototyping, RaPIER methodology and tool development will be guided by examples from actual Honeywell projects. The results of working the early examples will influence further RaPIER development. In addition, early versions of the RaPIER prototype engineering environment will be applied in practice projects at the research laboratory and in pilot projects at production sites. The final version of the RaPIER environment will be influenced by the results of these practice and pilot projects.

Figure 1-1 shows RaPIER's overall technical approach.



File No. 6-0283

Figure 1-1 Technical Approach to Developing a Prototype Engineering Environment

1.4 FIRST YEAR GOALS

RaPIER's first year goal was to lay the technical and methodological foundations for the prototype engineering environment, and to begin the technology transfer process. During this first year, the RaPIER project worked in five areas:

- o Software reusability. RaPIER prototypes will be developed with reusable Ada software parts. Reusable software possesses certain characteristics such as a flexible interface. Reusability characteristics are explicit goals for reusable software parts, and reusability guidelines are recipes for producing Ada software parts that meet those goals.
- o Software base technology. Prototype development with reusable parts will meet its speed and cost goals only if it is significantly faster and less costly to find software and reuse it than to recreate it. RaPIER's reusable software parts will reside in a software database. The software database management system must manage its contents so that prototype developers can find the "right" piece of software quickly. The way software parts are classified is a major factor in how fast they can be located, and how well they can be recognized as being the "right" software. Classification includes the specification of individual parts (for example, attributes such as author, number of source lines, and functional behavior), and a scheme that induces searching paths in a software repository.
- o Prototyping methodology. A prototype is a piece of software; like other software it must be designed and built in a systematic way. A prototype is used in investigating and clarifying requirements. This investigation should be carried out in a systematic way. Therefore the prototype must be built and used under the control of a methodology. A methodology is based on a life cycle; it constitutes a collection of methods to be applied at each stage of the life cycle, methods to accomplish the transition from one life cycle phase to the next, and ancillary methods such as a method for incorporating prototyping into the procurement process.
- o Computer automation of prototype building and use. Tools that automate the prototyping process will allow prototyping to meet its cost goal. Tools that support the prototyping methodology will ensure that prototyping meets its technical goal of requirements consensus. Since we believe that prototyping is the appropriate method for ensuring that a system meets its users' needs, we will develop our prototype engineering environment as a series of prototypes.
- o Technology Transfer. Technology is transferrable when it has recipients; the finest technical advance will not be applied in practice if it does not meet practitioners' perceived needs. Therefore practitioners should be R&D partners, expressing their needs and participating in developing the solution to their needs.

The RaPIER work is guided and influenced by experience gained from examples. Our first year plans included carrying out examples to validate initial conjectures about the abovementioned technology areas.

The next subsection describes the contract and parallel work we did to meet our first year goal.

1.5 FIRST YEAR OUTCOMES

Sections 2 through 10 of this report describe our work between January 1, 1985 and January 31, 1986. Each section presents the problem we attacked, the results we achieved, and the future work we plan. Section 11 presents a financial summary of the year's effort.

Section 2 provides background material which the RaPIER team needed to understand before undertaking the development of the RaPIER prototype engineering environment. It defines the broad application area, embedded computer systems, for which RaPIER prototypes will be developed, what constitutes a RaPIER prototype and who RaPIER's users will be. It also examines related work in the requirements and automatic programming areas which leads us to believe that RaPIER will be a novel system.

Sections 3 and 4 describe the methodology results achieved in this period. Section 3 presents three methodological elements which are the foundations of our future work: {1} constraints which a prototype and the prototyping process must meet, {2} a prototype engineering life cycle, and {3} a prototype construction methodology. The construction methodology is object-oriented; that is, it prescribes that prototypes be constructed from objects in the Smalltalk [INGALLS78] sense. Section 4 presents a model of computation that is appropriate for object-oriented program semantics, and suggests an implementation of the run-time support for this computational model.

Section 5 discusses requirements for a software base management system that can support programming with reusable parts, and a scheme for classifying those parts. The scheme is based on classifying software by logical behavior rather than actual routine or object names. Prototypes are written using logical behavior names, and logical behavior is bound to physical code only when a prototype program is compiled. This late binding allows software to be located in a software database undergoing frequent insertions and deletions.

Section 6 presents three software reusability meta-characteristics and fifteen reusability characteristics as the explicit basis for Ada reusability guidelines. It then presents a selection of reusability guidelines which software designers can apply to design and code reusable Ada software parts. These guidelines are recipes for achieving the characteristics. The guidelines are arranged in chapters which parallel the Ada Language Reference Manual [DOD83]. The section also contains sample Ada modules which exemplify the guidelines.

Honeywell practitioners have sent a loud, clear message that programming with reusable parts will not yield substantial time and cost saving if it does not provide some method of quickly developing new software parts when reusable parts are not available. Section 7 examines one alternative to line-by-line coding of new software. It presents a technology assessment of the current

state of the art in program development using program transformation systems such as DRACO [NEIGHBORS80], and concludes that this technology is either too immature or not targetted correctly for easy inclusion in the RaPIER environment.

To achieve our technology transfer goals, we must develop a prototype engineering environment that solves practitioners' perceived problems, and meets their requirements. To do this, and to educate ourselves about prototyping, RaPIER methodology and tool development is being guided by examples drawn from real Honeywell programs. During the past year we developed three prototypes: two of facets of Honeywell's Space Station work, and one of a fragment of the RaPIER environment. Section 8 reports on the Space Station example prototypes, and the lessons learned from developing them.

In addition, the RaPIER project has formed a RaPIER Technical Advisory Panel (RaPTAP) comprising ten engineers from various Honeywell divisions. RaPTAP periodically reviews our work and advises us on its relevance to Honeywell divisional problems and the likelihood that the divisions represented will accept the technical solutions we are devising. Individual RaPTAP members also provide advice on a one-to-one basis.

Prototypes are developed from initial requirements presented by customers. We developed just such a set of initial requirements for the RaPIER prototype engineering environment itself. Section 9 presents these requirements, and a critique of them based on our fragmentary prototype of RaPIER. These requirements will be improved next year as a result of using the more complete RaPIER that we will develop.

1.6 FUTURE WORK

Throughout the first year of the RaPIER project, we decided to attack some problems and postpone consideration of others. In addition, one year's work is often not enough to complete work even on a problem which is being considered. Each section of this final report gives the details of proposed future work in a particular area. The following list suggests general directions for future work.

Our overall future plan is to continuously improve our prototype of RaPIER with new methods and tools, and apply the evolving prototype to an increasingly challenging set of example problems supplied by Honeywell divisions.

We will expand RaPIER's methodological foundations to cover the other phases of the prototyping life cycle and the transitions between phases. We will work to fit prototyping into the general product development life cycle, and into the DoD procurement process.

To support prototype construction, we will continue reusability and software base work until developing prototypes with easily locatable, well understood software parts becomes standard operating procedure. We expect to complete

Final Scientific Report: RaPIER Project (Contract No. N00014-85-C-0666)

the Ada reusability guidelines next year, to apply them to building RaPIER's repository of reusable Ada parts, and to improve the guidelines and the parts as we gain experience. We expect that the software database management system will be under development and enhancement for the next four years.

We will provide automation for all the methods we develop with a series of more and more complete prototypes of RaPIER. The eventual RaPIER prototype will support all phases of the prototyping life cycle.

The culmination of the RaPIER project is pilot use of a prototype of the RaPIER engineering environment in Honeywell divisions. In preparation for this step, we will conduct a domain analysis and develop domain specific reusable software parts for the application area(s) in which the pilot projects are conducted.

Finally, we will revise the prototype of RaPIER based on the results of the pilot projects. That revised prototype will be ready to be made into a product quality prototype engineering environment.

The RaPIER team develops new technology only when there is none available in either the commercial or the research marketplace. We will continue to track these marketplaces throughout the program, adopting and adapting as much existing technology as we can. RaPIER is supported in part by the DoD STARS Initiative's Application Area whose main thrust is software reusability. We hope to use results from other Application Area projects whenever possible.

SECTION 2

BACKGROUND

We began the RaPIER project with a careful examination of our objectives and non-objectives. This examination helped us define who RaPIER's users would be. Knowing the proposed user community allowed us to propose RaPIER system requirements that would meet their particular needs. This examination also provided a definition of what a RaPIER prototype would be. That information allowed us to propose a methodset and toolset for developing that kind of prototype. Finally, we examined related work in the areas of building models of systems and quickly building software, both to ascertain what technology we could adopt for RaPIER, and to identify RaPIER's niche. This section is a summary of that background study and those initial decisions.

2.1 EMBEDDED COMPUTER SYSTEMS

An accepted definition of embedded computer system (ECS) is "systems [that] are embedded in larger systems whose primary purposes are not computation...Common examples of embedded systems are industrial process-control systems, flight-guidance systems,...radar tracking systems, [and] ballistic-missile-defense systems..." [ZAVE82]. The class of ECSs is important to DoD because it includes most mission critical computer systems. Commonly, ECSs are real-time systems with critical timing demands and intricate ordering constraints. They are often distributed and have complex interface with the system in which they are embedded. During development, ECS requirements undergo constant changes because of changes in their surrounding system. Requirements problems also arise because ECSs are developed using state of art technology with which few developers have experience.

We are now struggling with the question of how prototyping can be applied in investigating requirements for ECSs. We must identify the end-user of a truly embedded system, the user visible function that can be prototyped, and the experiments that can be conducted with these prototypes. In the interim, we have chosen the broadest possible interpretation of the ECS definition given above, and have built prototypes of user-interfaces to control systems. We chose user-interfaces for two reasons: {1} because it is obvious how a prototype of a user interface can be used to pin down end-user requirements and {2} because "the portions of [multi-function systems] for which it is most difficult to define requirements are those supporting the cognitive processes of the user" [RADC84]. These examples have given us the opportunity to test our tools and methods for building prototypes, to think about tools and

methods for using prototypes, and to show our prototyping concept to ECS engineers who will help us define how to use prototyping for truly embedded systems.

2.2 THE USER

The "user" mentioned throughout section 1 is, obviously, the ultimate user of the system under development. However, there are two classes of prospective users of the prototype engineering environment. One is the prototype developer, who may be a specialist in prototype development, or a system or software engineer involved in requirements investigation who develops the prototype as a tool. The other is the prototype user, who must be an expert in the domain of the system under development. This domain expert may be a customer, an end-user or his/her representative, a procurement specialist, an engineer representing a prime contractor, or a system engineer from the same organization as the builder of the prototype and/or the final system. In this paper, we distinguish between prototype developers and prototype users, but group all prototype users together under the word "user."

2.3 WHAT IS A RAPIER PROTOTYPE

A RaPIER prototype is a vehicle for investigating computer system and software requirements. It is a working model of some requirements of a system under development. Initially, it serves as a baseline for changes that will lead to final convergence on requirements. Eventually, it represents a requirements agreement between developers and customers. We expect a prototype to be used interactively in controlled experiments. We expect that each experiment will lead to changes in the prototype. These changes serve two purposes: to give users the sense that developers have understood and reacted to their needs (communication), and to provide a better vehicle for subsequent experiments.

Because a RaPIER prototype is used to investigate requirements rather than being an initial version of a product, it is enough that it be an incomplete model of the system under development, showing only those aspects of the system requirements that need investigation. The requirements analysis step of the prototyping life cycle (see subsection 3.4) determines which requirements are to be investigated. In order to model the aspects of interest adequately, the prototype may also model other parts of the system as an environment for the aspects of interest. [ZAVE82] (see subsection 3.9, Note A) and [BALZER79] argue that the environment in which a system operates should be part of the system's requirements specification. This argument applies equally well to a prototype, since the prototype serves as a requirements specification. [BALZER79] says:

"...The environment in which the system operates and with which it interacts must be specified. Fortunately, this merely necessitates recognizing that the environment is itself a system composed of interacting objects...which are by definition unalterable...In fact, the

only difference between the system and its environment is that the subsequent design and implementation will operate exclusively on the specification of the system."

In concept, a prototype's environment comprises {1} aspects of the system under development whose requirements are not being investigated and {2} the external environment of the system under development. Parts of both components may be modeled in software. Those environmental things which are modeled are considered part of the prototype, but not part of the "prototype proper." Objects in the prototype proper are constructed to be modified easily. That is, their code can be changed so as to alter the behavior they display. Objects in the prototype, but not in the prototype proper, are not constructed for modifiability. It is "fair" to expect to modify parts of the prototype proper; in fact, prototyping experiments are planned to elicit judgments and measures that will lead to changes in the prototype proper. It is not "fair" to expect to be able to modify parts of a computer simulation of a prototype's environment. Using a prototype means giving it inputs. To the extent that inputs constitute the environment in which a prototype runs, that environment will change. But the code that models the environment will not change between experiments, except in extreme circumstances.

A RaPIER prototype is not a "black-box" model of system requirements. It exhibits structure. That structure is intended to reflect the users' view of the partitioning of a system's behavior into modules, rather than the partitioning that appears in design and source code. Partitioning according to the user's view should lead to easily modifiable prototypes. (See subsection 3.6, "A Prototype Construction Methodology," for details.) We conjecture that most users mentally partition the same system's behavior into the same conceptual modules, and that a prototype builder can capture that partitioning. The RaPIER project will test both this assumption of uniqueness and whether there is a method for capturing that unique users' view. [ZAVE85] claims that "most formalisms introduce internal structure - if only to decompose complexity" (see subsection 3.9, Note B). One obvious method of capturing that users' view is to use the partitioning in terms of which initial requirements are written.

There are two possible architectures for prototypes: the eventual implementation architecture, which is susceptible to incremental development, and the users' view architecture, which supports easy modifiability but not other desirable implementation traits such as reliability or survivability. Because the prototype is partitioned differently from the eventual implementation partitioning of the system, it does not lend itself naturally to incremental enhancement into final product. It can either be abandoned or retained for prototyping system enhancements.

Because the prototype is partitioned differently from the eventual implementation partitioning of the system, it also does not lend itself naturally to investigating design trade-offs. Perhaps later in the project we can devise methods of using a prototype which is NOT structured as the implementation is structured for investigating performance of implementation designs.

2.4 RELATED WORK

This subsection compares the RaPIER work to other work which entails either building models of systems or quickly building software. The comparison considers the purpose of the model or software and the approach to producing it.

2.4.1 Simulation

Simulation and prototyping are similar activities. Their high level goal is the same: to investigate aspects of a proposed or existing system in order to improve it based on the results of that investigation. The stages of simulation model development described in [FRANTA77] are very similar to the phases of the prototyping life cycle presented in subsection 3.4. The process (or scenario) view of simulation presented in [FRANTA77] and RaPIER's object-oriented system model are also quite similar. However, simulation and prototyping are not the same.

Discrete or continuous simulation is the process of building and exercising an abstract model of some crucial parts of a system in order to obtain performance(1) information that is impractical to obtain by analytic or numeric solution of the model or from direct use or observation of the system. A simulation model is based on an implementation architecture. It is a stochastic model and the results of exercising it are statistical predictions of system performance.

Prototyping, in the RaPIER sense, is the process of building and exercising a working model of some crucial parts of a system in order to determine whether the model behaves (functionally) in such as way as to solve the problem that the system is intended to solve. A prototype is not necessarily structured as an eventual implementation of the system under study will be structured. A prototype is a functional model and the results of exercising it are judgments about the system's adequacy as a solution to the problem at hand. We do recognize that system performance is an important facet of system behavior, and that a poorly performing system certainly is not an adequate solution to the problem at hand. Therefore we hope that RaPIER prototypes will eventually be able to answer performance questions. But for the near future they will be used to deal with questions of functional behavior only.

Simulation and prototyping are complements. Prototyping is used to determine whether a proposed system solves the right problem; simulation is used to determine the performance of that system. Because the intent of simulation is to determine system performance, it does not generally require a "realistic" model or one with which users can interact. Simulation is often used to

(1) We mean performance in the broad sense of speed, reliability, availability, resource needs, in fact all system characteristics except functional behavior, which can vary with variations in a system's architectural or algorithmic design.

provide designers with information to make design trade-offs. Prototyping, on the other hand, deals with users' needs and, therefore, prototypes must be "realistic" and support user interaction. The aim of prototyping is to exploit users' application experience and expertise. In the sense we are using the words, a traditional flight simulator is a prototype not a simulation, because of its realism and provisions for user interaction.

Prototyping and RaPIER are useful during requirements analysis, before an implementation architecture has been designed. Simulation, and simulation systems such as Simula or GPSS, are useful during design and implementation, when the system's structure is proposed or known, and the performance of that structure needs to be determined.

2.4.2 Application Generators

Application generators (AGs) are one of the oldest tools for fast development of software. "Their original purpose was very narrow, typically being used to generate programs whose output is a series of reports (such as RPG). Later, they were extended so they could interface with an existing database, perform statistical operations, and display the results" [HOROWITZ84]. Present day, commercially available AGs include Nomad, Focus, Mantis and QEE. AGs are geared primarily to support data-intensive business application development. They provide a very high level, special purpose, user friendly language and are often used by end-users. Their method of producing programs is to instantiate templates that are tailored according to the user's specification. The methodology for developing an application with an AG is to specify a limited prototype of the application and incrementally extend or modify that prototype into the final version of the program.

The major differences between RaPIER and extant or proposed [HOROWITZ83] AGs are {1} their domain of use, and {2} the functionality they provide in that domain. If the domain for RaPIER prototypes was data-intensive business applications, and if the functions to be performed were data analysis, reporting, database interaction, and some special functions like financial modeling, then RaPIER would be superfluous. However, our intended domain is embedded computer systems, and we wish to model general functional behavior in the domain. No extant or planned AG has that target domain and breadth of functionality. The embedded computer system domain is broader, less well understood, and changing more quickly than the business domain. Those domain characteristics seem not to permit definition of a set of code templates that an end-user could instantiate by specifications in a user friendly, non-procedural language and which would provide the necessary range of functionality. Therefore we expect that RaPIER users will be software specialists at least in the near future.

What RaPIER can learn from AG work is that the embedded computer system domain is too broad a target for the RaPIER system. Application generators do not support general business programming. They support special areas such as report generation or financial modeling. The embedded computer system domain must be broken down into domains such as trainer systems, space station, avionics, and the like. Analysis of the narrower domains may lead to several

special purpose prototype development systems, each based on the initial RaPIER system, with which domain experts with little computer experience can develop prototypes. This could happen because domain experts in a narrower domain than embedded computer systems could use even a fairly complex prototype development language that contained their concepts and vocabulary along with a rich set of templates, to develop prototype programs without help from a computer specialist.

Once RaPIER can support prototype development by software engineers, the same computer science technology plus domain analysis might help us specialize our PSDL and populate our software repository with domain specific reusable parts, thus allowing domain specialists (for example, system analysts) to construct prototypes.

2.4.3 Requirements Specification Systems

In the traditional software development life cycle [BOOCH83a], code is written after (at least the first iteration of) requirements and design work are finished. Thus a working system is available for the first time during testing of the implementation. [ZAVE79] documents the problems with this traditional life cycle. Requirements specification systems support this traditional life cycle. These three commercially available systems represent the spectrum of requirements specification systems:

- o SADT [ROSS77]. An unautomated graphical language for describing systems plus a methodology for producing the descriptions. It provides no prototyping capability or support for computer management of requirements information.
- o PSL/PSA [TEICHROEW77]. A computer aided system for documenting requirements that creates and maintains a database, analyses requirements for completeness and consistency, and produces a variety of printed reports. It provides no prototyping capability.
- o SREM [ALFORD77]. A system for documenting and simulating requirements for real-time systems that maintains a database and produces some printed reports. It provides two types of discrete event simulators; neither creates a "realistic" model of (parts of) the system under study with which a user can interact.

Requirements specification systems are an improvement over unautomated requirements analysis by ad hoc techniques, but they do not address the problem of exploiting domain specialists' expertise in the early life cycle phases. Prototyping is intended to make a working model of (crucial parts of) a system available early enough to exploit domain specialists' expertise before coding begins. From these systems, RaPIER can learn requirements information management techniques. These techniques might contribute to the incorporation step of the prototyping life cycle presented in subsection 3.4.

2.4.4 Language Based Prototyping Approaches

Language based approaches provide prototyping either as a by-product (for example, GIST) or as a result of the nature of the language itself (for example, Lisp). RaPIER is intended as a prototyping system with linguistic and methodological support for prototype construction, prototype execution, requirements analysis, and incorporation of prototyping results into the product development life cycle. RaPIER and an appropriate language could be mutually reinforcing. Our present approach to developing prototypes is to employ reusable software parts and a language to describe how the parts are assembled into a prototype. If a language became available that provided the same broad functionality as can be provided by a large repository of parts, and in which prototypes could be specified at the same high level as they can be in PSDL, then the RaPIER automation that supports the methodology (as distinct from the automation for language processing and the software repository) could be used to support that language. Such an alternative approach might be considered if, for example, the RaPIER system were to be introduced into a group that was expert with an appropriate language.

There are a number of language based approaches to prototyping, including:

- o Executable Specification Languages (ESLs): A specification is a formal statement which fully describes the intended observable behavior (the "what") of some object without describing how that behavior is to be implemented. An executable specification (ES) is one that can be executed, either by transforming it into an executable program or by interpreting it directly. An executable specification language has an operational semantics that permits its specifications to be executed or interpreted in some manner. An executed specification is a prototype, but changes are made to this prototype by changing the ES. Therefore, once the prototype is approved by the customer, the ES (a formal specification) is a valid basis for design, design verification, and implementation. ESLs are a key component of an alternative software development life cycle called the "operational" approach to software development.

Gist [BALZER82], OBJ [GOGUEN79] and PAISley [ZAVE82,ZAVE84] are well known executable specification languages. Gist and PAISley are intended for the development of efficient software products through correctness preserving transformation of the high-level operational specification into an optimized program. OBJ is a language for writing and testing formal specifications. All three make prototyping available automatically, but since their main goal is correct specification or efficient programs, they have neither methodological support nor special tools for prototyping. However, their major drawback for our project is their level of detail. The well known resistance to formal detailed specification languages among software practitioners militates against adopting an ESL for prototype development at first. Rather, we chose an approach that combines a formal but very high level PSDL with reusable parts that may be specified informally. However, using a detailed and formal executable requirements specification language is probably the "right" long range solution, since the all important incorporation step then becomes automatic.

- o Very High Level Languages (VHLLs): These are general purpose, executable notations containing constructs that express high-level functions or concepts, and for which there is a language processor that implements these functions. VHLLs allow programmers to describe high-level functions clearly and succinctly, thereby limiting source code complexity. Their processors implement the high-level functions or concepts, thereby freeing programmers from focusing energy on implementing them. VHLLs are not domain specific and usually "rely on a small number of semantically neutral primitive constructs such as mathematical sets" [BIGGERSTAFF84]. They sacrifice efficient execution for clarity and succinctness; that trade-off is acceptable in a prototyping language. SETL is a well known VHLL and has been used to prototype an Ada interpreter [KRUCHTEN84].
- o Problem Oriented Languages (POLs): The POLs of interest for prototyping are special purpose VHLLs that offer high-level primitives that are specific to a problem domain. They have all the advantages of general purpose VHLLs and offer concepts from the problem domain that can make source code programs even more succinct and lucid. POLs can, perhaps, permit application area experts to develop prototypes without help from software specialists. Some examples are: OPS5 [BROWNSTON85] in the expert system domain; MODEL [CHENG84] in the database domain; Gambit [LARRABEE84] in the video game domain; and PHINIX [BARSTOW85a] in the oil exploration domain. Some POLs are application generator languages, some use AI to aid program development, and some depend on traditional language processors. Whatever the means of implementing a program written in the language, the effect for the program writer is the same: a familiar set of concepts and vocabulary that hide low- and middle-level implementation details and permit the fast development of applications. These qualities support prototyping.
- o Flexible Languages: By flexible languages we mean interpreted languages such as Lisp, Lisp with Flavors, SMALLTALK, or APL, that give prototype developers an interactive environment in which to build prototypes incrementally, and which produce an easily modifiable prototype. Prototyping is a kind of exploratory programming and an initial prototype is a sketch. It is a "given" that a prototype will change both during its development and while experiments with it are in progress. Under exploratory conditions, it is easier to develop a prototype in a language which allows "the programmer to defer commitment as long as possible. A decision that has not been cast into code does not have to be recast when it is changed. Thus, the longer one can carry such a decision implicitly, the better" [SHEIL83a].

In addition to allowing deferred commitment, the languages cited above provide "an integrated programming environment which uses language specific programming knowledge to provide exactly those control and bookkeeping functions that are the greatest drain on a programmer during rapid system development" [SHEIL83a]. Prototyping is, of course, one kind of rapid system development.

The drawback of flexible languages is that their users must be "gurus." No end-user will ever develop a prototype in Lisp, nor will a software engineer who doesn't know Lisp. However, for prototypers who are proficient in Lisp or SMALLTALK, that language along with its environment and a library of reusable objects may be an ideal foundation for RaPIER.

2.4.5 Incremental Development

It is tempting to consider developing a product in stages, providing early versions (prototypes) for users to judge and enhancing those early prototypes into a final product. However, if prototyping is undertaken for requirements clarification, this can be a dangerous strategy for other than small programs that will be used only by friendly users. According to [GOMAA83]

"with small systems, it may be feasible to allow a prototype to evolve into the production system. However, with larger systems, this is unlikely to be advisable because the software engineering approach required to develop a production system is very different from that required to develop a prototype."

And, in likening prototypes to scale models, [WEISER82] says

"A prototype is one kind of scale model...A scale model provides scalable (that is, generalizable) information about a system's actual interaction with its intended environment...A quickly built software system which is equivalent in every way to the desired end product is not a scale model but just a cheap final product. A scale model sacrifices something for its speed of construction. (FOOTNOTE: Too often what is sacrificed is maintainability, and many a scale model has been sold as a final product with disastrous effects for maintenance.)"

If prototyping is undertaken to clarify requirements, we recommend building a "throw away" prototype in the interest of obtaining a robust final system that is built to meet qualitative requirements, such as maintainability.

blank back page

SECTION 3

PROTOTYPING METHODOLOGY

This section describes initial work on the methodological underpinnings of the RaPIER prototype development environment. The section deals with two major elements of prototyping methodology: a prototyping life cycle and a prototype construction methodology.

The material presented here is the result of work on Task H1.6 and Honeywell funded work.

3.1 PROBLEM STATEMENT

Prototyping is not hacking. A prototype is a piece of software; like other software it must be designed and built in a rational way. Like other software it must exhibit certain qualities, modifiability for one. A prototype must be built quickly. A prototype is used to achieve a goal: identification and clarification of requirements. The requirements under investigation must have been selected in a rational way. Finally, the results of exercising a prototype must be systematically incorporated into the product development process. These needs will be met only by building and using prototypes under the control of a methodology.

3.2 OUTCOME

We started the methodology task by defining three elements which are the foundations of further work: {1} constraints which a prototype and the prototyping process must meet, {2} a prototyping life cycle, and {3} a prototype construction methodology. Explicit constraints on prototypes and the prototyping process provide a yardstick against which to measure methods and their supporting tools. The life cycle serves as a roadmap for the rest of the methodology work and for the entire project. The construction methodology provides needed guidance for developing prototypes. This guidance is important because our research methodology is based on building example prototypes to direct the course of the development of both the RaPIER tools and the execution methodology.

3.3 CONSTRAINTS

We have identified four constraints on the prototyping process and on the prototype itself. Three were mentioned in subsection 1.3.

1. The process must be relatively inexpensive. A development project should dedicate resources to prototyping equal to what can be saved by eliminating risks during development and eliminating rework immediately following initial product release. [GOMAA83] argues that the prototyping life cycle costs must be no more than ten percent of the whole development cost. It is difficult to measure whether ten percent of development resources is the amount prototyping saves, less, or more. The ten percent figure also does not include reported user satisfaction with the installed system [ALAVI84] caused by early "buy-in" to development goals and decisions. However, ten percent seems to be what the traffic will bear.
2. The process must be relatively quick. The purpose of requirements prototyping is to get the kind of feedback from customers that is obtainable only when they have hands-on experience with a working version of (a part of) the system. If this feedback is not obtained quickly, it cannot influence requirements analysis. Speed, of course, is relative to the length of the entire development: a six month project needs a prototype within the first month while a six year project can wait six months.
3. The prototype must be a communication device. Traditional black-box requirements are difficult to discuss even among computer specialists, but especially between domain experts who are not computer scientists and the computer scientists who are solving their problems. [ZAVE85] states that "Another important factor in user/analyst communication is the ability of the user to grasp and evaluate the concepts behind any proposal. Experienced systems analysts report that an explicit operational model is much more helpful than black-box requirements....." That structure will model the user's view of the solution to the problem, not the structure of the eventual implementation of the solution (see subsection 3.9, Note B). The CEO of Black & Decker is reputed to have said that his customers don't want drills, they want holes. The prototype must show holes, not drills.
4. The prototype must be modifiable. The purpose of experimenting with a prototype is to generate common understanding and comments that lead to changes in the system requirements. The "experiment-modify" cycle necessitates changing the prototype to give users an updated discussion vehicle and to demonstrate developers' responsiveness.

The RaPIER methodology supports the construction and use of the sort of prototype described here, under these process constraints.

3.4 A PROTOTYPING LIFE CYCLE

The RaPIER project has proposed a four phase prototyping life cycle:

1. Analyse the system's (or system software's) initial requirements. Determine which are poorly understood, high risk, possibly incomplete, have caused problems in past systems, or represent new technology. Formulate questions about them that can be answered by exercising a prototype. These questions are part of the input to the design-and-build phase of the life cycle.

The methodological support for this phase comprises:

- o criteria for deciding which requirements to investigate;
- o methods for formulating questions about the requirements under investigation;
- o a strategy for designing experiments that will get answers to the requirements questions.

2. Design and build a software prototype to answer the requirements questions generated in step 1. This prototype may model only selected parts of the system under development, and may model some of the environment of the system (or parts) under study (see subsection 3.9, Note A and the discussion in subsection 2.3).

The methodological support for this phase comprises:

- o a model of prototypes in terms of which developers design prototypes (see subsection 3.5);
- o techniques and a language for constructing prototypes according to the model;
- o techniques for ensuring that the prototyping process and the prototype itself meet the criteria set out in subsection 3.3.

3. Experiment with the prototype, modifying it in response to domain experts' comments. These experiments both foster communication between system or software developers and domain experts and produce changes in the baseline requirements. Modifying the prototype may involve repeating portions of phases 1 and 2 of the life cycle.

The methodological support for this phase comprises:

- o procedures for systematic experimentation with prototypes;
- o criteria for deciding how long to continue prototyping experiments and how extensive each experiment must be;
- o a notation for stating the results of the prototyping experiments, that is, stating the prototype's behavior. This must eventually be a formal notation if step 4 of this life cycle is to be carried out formally.

4. Incorporate the results of the prototyping experiments into the final requirements. These final requirements, the engineering response to the initial requirements, are used in designing and building the actual system.

The methodological support for this phase comprises:

- o a formal notation for specifying the prototype's behavior, as input to the incorporation process;
- o a mapping from that formalism into the formalism of the engineering response document;
- o a formal notation for stating the final requirements.

Because a prototype is a program, it formally represents the requirements agreements between customers and vendors. That program is made up of code units with (semi-)formal specifications and composition operators (see subsection 1.3) with known semantics. These formal objects--program, code unit specifications, and composition operator semantics--should be usable to generate a formal engineering response through formal transformations. There are two factors militating against using a formal incorporation approach in the near future:

- o lack of a common use of formal specifications for describing the behavior of individual code units;
- o lack of a common use of formal specifications for expressing system requirements.

Although formal requirements languages exist (for example, Gist [BALZER82], PAISley [ZAVE82], and RSL [ALFORD77]), we feel it is dangerously speculative at present to choose one and develop a mapping to it from some formal description of the prototype's behavior. The work needed to make the selection and develop the mappings is beyond the scope of the RaPIER project. Therefore the project can, at best, recognize the problem and suggest an approach to solving it.

Subsection 3.6 describes the methodology for the Design and Build phase of the prototyping life cycle in detail. Methodological support for the other life cycle phases is part of the RaPIER project's future work.

3.5 THE PROTOTYPE MODEL

The model of prototypes characterizes a prototype's conceptual architecture and the abstract building blocks that populate the architecture. The model constrains the ways an informal requirements specification will be realized as a prototype to those that can occur in the conceptual architecture with the conceptual building blocks. Prototype builders think in terms of this model when putting prototypes together. This model is formulated in computer science terms; it is not necessarily the model that prototype users envision when interacting with the prototype.

A prototype is a collection of objects which operate concurrently and autonomously. The objects communicate asynchronously by passing messages. Internally, each object may contain some local state and sequential procedures (or methods) with which to respond to messages. This model is similar to SMALLTALK's [GOLDBERG83] model, except that inheritance is not currently a feature of RaPIER objects. We are not ruling out inheritance as a characteristic of RaPIER objects. We simply have not yet investigated the interaction between inheritance and other RaPIER assumptions, for example the assumption that RaPIER objects will most likely be implemented in Ada. Our model is also similar to the Gist [BALZER79], CSDL [WOOD84] and Flavors [CANNON82] models. Section 4 discusses one implementation of this conceptual architecture.

This model was chosen for several reasons:

- o Objects (rather than subroutines, data structures, or general code fragments) are a convenient unit of reuse. The work on abstract data types [LISKOV75], SMALLTALK [GOLDBERG83], and Flavors [CANNON82] bears this out. Prototype developers using RaPIER will think in terms of combining reusable units. These units should present complete enough behavior to be {1} understood and used as units rather than incomplete fragments, and {2} combined without internal modification.
- o If prototype building is to be carried out at a higher level than line-by-line coding, it will have to be carried out in a very high level language (VHLL) that suppresses details of the running prototype. If that language supports a model or paradigm, programs written in it can be concise and still clear because mechanisms implied by the paradigm do not have to be stated explicitly in the program. We think that a VHLL for prototyping must suppress control details by supporting a paradigm that provides a control metaphor. We also think that information flow, in this case in the form of messages, is an appropriate metaphor for control in a VHLL for prototyping. When people draw informal pictures of systems, they often draw objects and information flow among them, suppressing other details such as control flow. Our adoption of the object/message paradigm provides a prototype control metaphor that developers appear comfortable with when describing the kinds of systems whose requirements will be investigated by prototyping [ZAVE82]. We conjecture, therefore, that a prototyping VHLL that supports the object/message paradigm will be natural to learn and use.
- o We conjecture that users interacting with a prototype will view it as a collection of autonomous, concurrent processes. The builders' model, which uses objects to implement concurrent processes, was chosen in part because of this supposed coincidence of views. Although users will not think in computer science terms, of objects with local state and methods, and of asynchronous communication by message passing, they will think of a collection of processes, modules, or objects, each responsible for some part of the prototype's behavior. If the objects with which the builder works are the same as the objects the user conceptualizes, changes to the prototype will be localized, yielding an easily modifiable prototype. The next subsection examines this reasoning in detail.

3.6 A PROTOTYPE CONSTRUCTION METHODOLOGY

This year the RaPIER project concentrated its methodology work on prototype construction, the second phase of the prototyping life cycle. This paper presents RaPIER's proposed methodology for quickly and inexpensively building modifiable prototypes that serve as communication devices during requirements analysis. The methodology is operational [ZAVE82] and object-oriented [BOOCH83].

A prototype is a "white-box," operational requirements specification. "An operational specification is a[n]...executable representation of [a part of] the proposed system. It is described in terms of computational structures that are known to have a wide range of possible implementations, and its organization is based as closely as possible on the problem to be solved" [ZAVE85]. The computational structures are modules that, in concept, operate concurrently and communicate by message passing. Messages are data or control requests. The modules realize problem oriented behavior; they reflect the user's view of a proposed system.

The modules are objects in the SMALLTALK [INGALLS78] sense: autonomous loci of control that interpret the messages sent to them by internally defined methods. At present we have not included the inheritance mechanism in our model of objects. Without inheritance, these objects are similar to those produced using the object-oriented design methodology described in [BOOCH83]. They are "black-boxes" inside a "white-box" structure.

The prototype is expressed in a Prototype System Description Language (PSDL) whose "nouns" are the above-mentioned objects, and whose "verbs" are composition operators that determine the control regime among the nouns. The threads of control in the running prototype are determined by {1} the composition of the entire prototype, {2} the messages objects send to one another, and {3} each object's internal control structure.

In summary, our methodology is operational in that it uses a prototype description language that can be executed and that deals with internal structure. It is object-oriented because the internal structure is composed of objects. The operational prototype description is represented in the PSDL notation.

3.6.1 The Operational Approach

3.6.1.1 Explanation and Justification

The operational requirements definition approach [ZAVE85] assumes that a user's informal statement of needs is formalized as a "white-box" operational specification, giving a problem oriented architectural model of the desired system.

The operational approach, as presented in [ZAVE82], "... is to specify the requirements for an embedded system with an explicit model of the proposed system interacting with an explicit model of the system's environment... The entire model is executable." The approach is operational "... because the emphasis [is] on constructing an operating model of the system functioning in its environment." [ZAVE82] contains detailed justifications of the approach.

[ZAVE85] presents an operational approach to the entire software development process. The initial phases of the operational approach (problem understanding and [requirements] specification) are explained as follows:

"During the specification phase, computer specialists formulate a system to solve the problem and specify this system in terms of implementation-independent structures that generate the behavior of the specified system. The operational specification is executable by a suitable interpreter. Thus external behavior is implicit in the specification (but can be brought out by the interpreter) while internal structure is explicit.

"This description may make an operational specification sound like a design, but it is not. First of all the structures provided by an operational specification language are independent of specific resource configurations or resource allocation strategies (and can be implemented by a wide range of them), while designs actually refer to specific runtime environments....

"Not only are the structures of an operational specification language independent of implementation-oriented decisions, but also the mechanisms (usages of the specification-language structures) in an operational specification are derived solely from the problem to be solved. They are chosen for modifiability and human comprehension without regard to any implementation characteristics whatsoever."

In developing software, the problem-oriented requirements specification is transformed into a solution-oriented design and implementation. In developing a prototype, the problem-oriented structure is retained, and the specification is transformed into a running prototype.

We conjecture that retaining the problem oriented structure will make the prototype a comfortable communication vehicle, meeting constraint 3 set out in subsection 2.3. Though the fact that the prototype has a modular structure will not be apparent in the running prototype, we think that users will discuss the prototype's behavior in chunks that correspond to their view of the system structure. We also believe that the users' chunks will correspond to the prototype's module structure, meaning that the chunks of behavior under discussion will be realized as one or a few related modules. These modules are chunks that the prototype developer, a computer specialist, also understands as units. These modules then, confine and direct discussions.

The prototype will be represented in a Prototype System Description Language (PSDL) [FRANKOWSKI85]. PSDL's "nouns" are problem-oriented objects; its "verbs" are operators which compose multiple objects into one source code

program which is then translated into an executable program. The preferred way to modify a prototype is to change its PSDL representation. These changes will cause objects to be removed from the prototype, new objects to be included in it, or changes to be made in the overall architecture or control regime. The fact that objects are autonomous entities with intrinsic behavior allows changes to be made by the simple inclusion/exclusion of entire objects. Some finer grained changes may be made by modifying the objects that comprise the prototype. Retaining the prototype's problem-oriented structure makes these kinds of modifications manageable. Since the user's model of the system is in terms of these objects, his/her comments about the prototype, and suggestions for changes, will also be in terms of these objects. In most cases, suggested changes should be local to one or a few modules. Either an entire object will be included in/excluded from the prototype, or parts of a module will be modified. In any case, changes will not be diffused among many modules.

3.6.1.2 Implementation

The transformation that turns an operational requirements specification into a running prototype consists of assembling the reusable building blocks referred to in the operational specification into a running program. Those building blocks are objects, residing in a software base. They are stored according to the classification scheme described in [ONUEGBE85], and retrieved under the control of a PSDL which contains the operators that glue the objects into a running program.

The RaPIER execution environment [FRANKOWSKI85] executes the prototype. The running prototype will eventually be a mix of compiled and interpreted code. Modifications during prototype exercise will be made either by changing the PSDL text, which will then be reinterpreted, or by dynamically linking a new version of some object into the prototype. Suspension with state saving will allow prototype runs to be resumed from the point of change rather than restarted.

3.6.2 The Object-Oriented Approach

3.6.2.1 Explanation and Justification

Object-oriented programming is the programming paradigm embodied in SMALLTALK [GOLDBERG83] and the Lisp flavor system [CANNON82]. The concept of an object as a named computational entity with an identifiable behavior is central to object-oriented programming. An object's behavior is its reactions to the set of messages it "understands," where a message is a request to initiate processing or provide information, and "understanding" means possessing a defined response. Messages are something like conventional procedure calls, with the important distinction that "a conventional procedure call ... denotes an action, and sending a message ... makes a request. In a typical procedural [regime] it is hard to give up the notion that the caller of a procedure is somehow 'in control.' ... In [the object-oriented world], on the

other hand, a message is a request of what the sender wants with no hint or concern as to what the receiver should do to accomodate the sender's wishes. The sender, presuming all objects to be quite intelligent, trusts the receiver to do 'the right thing' without worrying about exactly what the right thing is. Thus assured, the sender relinquishes control philosophically as well as actually, so that the interpretation of the message is left entirely up to its recipient." [RENTSCH82].

Object-oriented programming can be carried out top-down or bottom-up. Top down object-oriented programming comprises six activites [BOOCH83]:

1. Define an informal strategy for solving the problem at hand (in our case the informal strategy is the result of requirements analysis),
2. Identify the objects (nouns) in the informal strategy,
3. Identify each object's operations (verbs) in the informal strategy,
4. Define each object's interface; that is, the services and information it offers to other objects,
5. Implement each of the objects,
6. Implement the informal strategy as a program that uses these objects.

In this case an implementation is developed for each object needed.

Bottom-up object-oriented programming begins with a collection of reusable software objects such as the SMALLTALK system's objects, the collection of flavors that comprise a Symbolics operating system [SYMBOLICS84], or a user's personal library. Objects for the problem at hand are built up by combining more primitive (system or user-defined) objects. Eventually the system contains the appropriate objects to solve the problem at hand. Then a program is written that uses these objects. Bottom-up object-oriented programming is a natural way to exploit a software repository's resources. The program under construction can certainly be designed top-down, but that design will take into account the available resources and build down to meet them.

Our construction methodology is designed to produce prototypes which meet constraints 3 and 4 of subsection 3.3 by a process which meets constraints 1 and 2 presented in the same subsection. Our work in the coming years will test whether the methodology does this.

Ready-to-use building blocks of any sort for prototypes ensures the quick construction of inexpensive prototypes, meeting constraints 1 and 2. Defining these building blocks to be objects which encapsulate their own behavior ensures easy modifiability of an individual object and of the entire prototype, meeting constraint 4. The objects we propose are user oriented; they appeal to the user's intuition about the system he/she is cooperating in specifying. We expect, therefore, that comments about the prototype will naturally localize themselves to one, or a few, object(s) at a time. This also localizes the modifications needed to show that the developer understood

the user's reaction to the prototype. One object can be replaced by another which understands the same messages, but reacts differently to them, or methods within an object can be replaced by others that implement different behavior. Modifying individual objects results in modification of the entire prototype.

Another major value we think results from object orientation, rather than, say, functional orientation, is that the objects which implement the prototype match, or come close to, the user's intuition about the system's structure. This makes the prototype a comfortable communication vehicle, meeting constraint 3. In addition, the object paradigm controls the complexity of the development process in the same way that abstract data types do, by providing prototype developers with high level abstractions that are meaningful in the application domain of the system under study. Yet another benefit of object oriented prototypes may be that properly designed objects may also be usable in a final system or at least serve as detailed specifications for objects in the final system (see subsection 3.9, Note C).

3.6.2.2 Implementation

The RaPIER prototype engineering environment will contain a software database [ONUEGBE85] of objects designed to support prototyping in particular application areas. The collection of objects will comprise some general purpose objects, such as a window manager, a graphics package, or a text editor, and application specific objects. For example, a RaPIER environment for avionics prototyping might contain specific objects such as an air speed sensor or a flaps manipulator. These objects are intended to be reused in many prototypes. We expect that most of the objects in the RaPIER environment will have been developed by the RaPIER team and other Honeywell personnel. However, the database system will be able to absorb software from any source. As long as a software module is an object in the sense we have been discussing, or can be made into one, we want to acquire it for reuse. [ONUEGBE85] discusses the management of large software repositories.

Initially, at least, there will not be a reusable module to realize every behavior a prototype must exhibit. As more prototypes are developed, however, the stock of reusable modules will grow. Barstow's [BARSTOW85] experience in developing a rule base for transformational programming showed that "[a]s successively harder programs were attacked during the process of rule development, fewer and fewer rules needed to be added to the knowledge base. And when a new domain ... was tackled, it became clear that much of the necessary knowledge was already covered ... the process of developing rules for a given task was considerably simplified by the fact that much of the necessary knowledge had already been codified for rules in other tasks." This experience will most likely be duplicated for reusable objects.

When a reusable module is not available to implement some needed behavior there are two choices: build the module from scratch or use some function of a complete program such as a text editor, spreadsheet, graphics package or compiler. New components may be "hand crafted" according to the reusability guidelines, inserted into the software base and extracted under software base

control. However, for a prototype of a state-of-the-art system, there will be many missing functions and hand crafting each one will cause a bottleneck. Therefore, we are looking into providing new modules by means of executable specifications and/or application generators.

3.6.3 The Construction Procedure

Like any program, a prototype is constructed by building subsystems and putting them together into larger subsystems until the prototype is complete. We recommend these steps, in this order, for constructing each piece of, and the entire, prototype:

1. Understand the requirement(s) that is/are to be represented. The requirements analysis phase of the prototyping life cycle will have produced a set of requirements questions. The design phase begins with understanding them and devising a method of modeling them.
2. In a manner more problem-driven than parts-driven, define a set of objects that will represent the behavior of the requirements under study and the communication paths among them. Express the objects' definitions in some (eventually formal) specification language. Express the prototype's architecture in PSDL as an operational, object-oriented specification.

Design should proceed in a net problem driven manner. By this we mean that the definition of the objects that will comprise the prototype is influenced more by what is needed to represent behavior than by what is available in the software repository. However, the availability of reusable parts must influence the definition of the objects, otherwise having a parts repository will not speed up the construction process at all. We expect some depth first exploration of the implementation of a certain behavior with available parts, and backup to redesign objects when a useful but not "exactly right" object is found in the repository.

3. Combine the parts into an initial implementation using PSDL. Submit the PSDL to the language processor for synthesis and execution. Note that we expect PSDL to serve as both design and implementation notation. PSDL provides primitives to realize an object-composition paradigm, allowing prototype developers to express implementations at a very high level, almost a design level. The PSDL language processor allows such a high level notation to be executed.
4. Repeat:
 - o search for objects in the software repository,
 - o modify the prototype design and its PSDL specification to include or exclude objects
 - o create new objects if necessary
 - o modify objects internally only if absolutely necessary
 - o submit the prototype specification to the PSDL processor for synthesis and execution

until the PSDL specification of the prototype is satisfactory.

Experiments conducted during phase three of the prototyping life cycle will inevitably lead to modification of the prototype. Modification will be similar to initial building, except that only portions of the prototype will be rebuilt. The rebuilt portions will be integrated into a prototype execution at a suspension point. Resynthesis of the entire prototype and restart of execution will not be necessary. Building and modifying prototypes is exploratory programming [SHEIL83] both because prototype requirements will change as construction continues and because the implementation approach of reusing parts necessitates design changes as parts are found. Therefore, we expect the process to be replete with backtracking. [SHEIL83a] deals with the language and run-time support features necessary to facilitate exploratory programming.

3.7 EXPERIENCE

During the past year we built two example prototypes; they are described in subsection 8 of this report. Both were built following our proposed prototyping lifecycle and construction methodology as much as possible. These are some lessons we learned from that work:

- o The proposed prototyping lifecycle appears to describe the process in a useful way. The traditional waterfall description of the software development lifecycle [BOEHM83, HAMILTON83] is criticized because, although it shows iterations between stages, it does not adequately reflect the interleaving of stages. We have discovered that our four phases describe distinct activities, not interleaved activities, and that these activities are conducted in the order proposed. We conclude that we have partitioned the prototyping lifecycle into big enough chunks, and that those chunks cover the prototyping process. Therefore one can manage a prototyping project successfully using the proposed lifecycle. In addition, this lifecycle is an appropriate basis for future methodology work.
- o White-box specification of prototypes is useful for both prototype builders and prototype users. Neither builders nor users see the prototype as a black-box monolith; both see it as possessing internal structure. More importantly, users appear to see a hierarchical structure, but with components at various levels in the hierarchy visible at the interface between themselves and the prototype. When builders use a white-box specification technique, they apparently can design a prototype that shows users a structure that matches the way those users want to discuss the system during prototype exercise.
- o Objects appear to be an appropriate paradigm for the prototype's components. Builders of prototypes, as opposed to product software that must perform efficiently, apparently think in terms of objects. When objects are available for prototype construction, the translation of the structure in a builder's head to the prototype program's structure is an identity transformation, which requires less effort to do than a more complicated transformation. In addition, when builders capture users'

concept of the structure of the system under study, users and builders do discuss the same set of objects. This localizes a change to one or very few objects in the prototype, resulting in easily modified prototypes.

Section 8 contains the details that substantiate these tentative conclusions.

3.8 FUTURE WORK

A prototyping life cycle implies a collection of methods for each phase of that life cycle and for transitions between phases. Here we present some of the tasks that remain to be done in developing that complete collection of methods, in the order in which we will attack them:

- o Construction Methodology: Our proposal for a prototype construction methodology combines an operational approach to requirements specification that is "... relatively new and untried..." [ZAVE85] with an object orientation that has succeeded in development methodologies ([BOOCH83], the Symbolics environment) but has not yet been tested for requirements specification and investigation. In particular, subsection 2.3 argues that most people mentally partition a system's behavior into the same conceptual modules, and that a prototype builder can capture that unique partitioning. This uniqueness conjecture was influential in selecting an operational approach, and must be tested extensively. Some initial testing leads us to believe that the uniqueness conjecture is too strong, but that the builders and users of any particular prototype can agree on a partitioning. The question of whether there will always be objects in a software base to realize that partitioning must be examined carefully.

Our task, then, is to validate that this construction methodology is appropriate for prototyping. We plan to accomplish the task by applying the methodology immediately in constructing prototypes of the RaPIER prototype engineering environment and some application systems. These tests will validate whether the idea of an operational requirements specification is as useful as Zave claims and as our survey of requirements documents produced in Honeywell indicates, and whether objects are the correct constituents of such operational specifications. We are certain that this methodology needs extensive refinement. Our practice with the methodology will lead to those refinements.

We have not considered inheritance in our object model. We must investigate whether we need the inheritance component of the object model and, if we do, how to implement it in Ada with the aid of a software base management system.

- o Execution Methodology: Our next concern is the execution methodology. Our task is to develop procedures for insuring that prototype execution is a process of systematic experimentation rather than hacking. We suspect that systematic experimentation will be facilitated by scripts of the necessary interactions between a user and a prototype. We must validate that conjecture and, if it is true, develop principles for developing such

scripts. If the conjecture is not true, we must propose another prototype execution approach and develop methods for carrying it out. We must also develop an appropriate set of measures for ensuring that prototype use leads to answers to the questions posed during requirements analysis.

- o Experiment Design: If prototype execution is to be a systematic experiment rather than hacking, the experiment must be designed. We know that experiment design affects prototype construction, since the prototype must be modifiable in those areas where experiment will take place. Experiment design affects prototype execution in that the execution will be controlled by the experiment. We think that requirements analysis and experiment design should be conducted at the same time, since the experiment will test those aspects of the requirements which were questioned during requirements analysis. Our tasks are to develop experiment design techniques and determine when experiment design should occur. Experiment design techniques includes criteria for knowing when to stop the experiments.

Methodology for the requirements analysis and incorporation phases of the life cycle will be considered further in the future.

Incorporating Prototyping Results: The "burning question" in the incorporation phase is mapping from the formalism of the prototype to the formalism of the engineering response. Work on this question is outside RaPIER's current scope, for technical and funding reasons explained in subsection 3.4. Therefore, we will develop some informal methods for assuring that the results of the prototype experiments are reflected in the final engineering response. These may be checklist techniques. Technical considerations militate against developing a formal mapping at this time; however, an approach to a formal mapping might be developed.

- o Requirements Analysis: On the one hand, deciding which requirements to investigate by prototyping is just another case of the general problem of deciding which requirements are not well enough understood to proceed with design. Our task here is to monitor general work in requirements analysis and adopt criteria and methods from that work. On the other hand, requirements analysis prior to prototyping is also concerned with deciding which requirements can be profitably prototyped; that is, which requirements will developer and user be better able to discuss with the help of an animation. Our task here is to develop an approach to answering this question.

Here are some miscellaneous methodology problems which must be solved before the RaPIER system can truly meet its goals:

- o Domain Specificity: We believe that rapid prototyping is easier to achieve in a domain specific prototype engineering environment than is a general purpose prototype engineering environment [NEIGHBORS80]. To test this, we must select a domain for practice and pilot projects, conduct a domain analysis, and, based on that analysis, develop or acquire a collection of domain specific reusable software parts. We must also identify the domain analysis techniques necessary to transfer the RaPIER prototype engineering environment to a new domain.

- o Prototyping in the Development Life Cycle: Prototyping is not a "stand alone" activity. It supports product design and development, for which there are many accepted life cycle models; and it occurs in some procurement process. Our task is to suggest ways to fit prototyping into acquisition processes and general development lifecycles.
- o Change Management: Changes in the surrounding system which induce changes in ECS requirements are a major problem in developing embedded computer systems (ECSs). These changes are likely to happen throughout the development life cycle. Changing requirements is also a problem in any computer system development effort. A requirements engineering methodology must manage requirements changes. Some change management issues we must investigate are: {1} investigating requirements changes with a prototype that was not built with those requirements in mind, {2} propagating changes to the design and implementation phases, {3} distinguishing requirements which are likely to change by some special incorporation procedure, and {4} tracking changes.

3.9 END NOTES

- A. "Including an explicit model of the environment has several advantages for requirements specification. The reason that the interface between an embedded system and its environment is complex, asynchronous, highly parallel, and distributed is that it consists of interactions among a number of objects which exist in parallel, at different places, and are not synchronized with one another. Organizing these interactions around the objects (processes) which take part in them is an effective way to decompose this sort of complexity. Furthermore, assumptions and expectations on both sides of the boundary can be documented. The result is a specification which is far more precise and yet comprehensible than could be obtained by treating either side of the interface as a "black box," which is what happens when the environment is not modeled." [ZAVE82].
- B. In defending the proposition that operational requirements specification is an implementation-independent, "white-box" structure instead of a "black-box," [ZAVE85] states:

"The conventional approach stresses that all behavioral decisions should be made before any structural ones. This is an unrealistic and even undesirable expectation, since internal structure inevitably affects such external properties as feasibility, capacity, behavior under stress, and interleaving of independent events. Previous examples have illustrated this...

"Software can be made fault-tolerant only if it has meaningful components whose failure can be detected and whose bad effects can be contained. To the extent that fault tolerance involves user participation, external behavior cannot be analyzed or defined without a virtual structure of components to which fault-tolerant properties can be attached.

"Even if we could develop adequate behavioral requirements free of structural bias, there would be considerable difficulty in specifying them formally, since most formalisms introduce internal structure - if only to decompose complexity. Sets of axioms and finite state machines... have proven useful only for specifying components of complex systems [or] only for specifying very limited properties of complex systems. This problem is undoubtedly part of the reason why most conventional requirements are still written in English.

"Another serious problem with the conventional approach is its reliance on a strategy of top-down decomposition for design. Basic methodological principles tell us that implicit decisions should be avoided, that if error-prone decisions must be made early then they should be subjected to early checks, and that individual decisions should be as orthogonal to others as possible. Top-down design leads to decomposition decisions most of whose consequences are implicit, makes the most global decisions earliest yet cannot validate them until the very end, and causes the top-level decisions to affect all properties of the system. It seems that top-down hierarchical decomposition is an excellent way to explain something that is already understood but a poor way to acquire understanding...

"In the operational approach the primary decomposition of complexity is based on problem-oriented vs. implementation-oriented structure rather than hierarchical decomposition. Even within an operational specification, the most prominent structures tend to be discovered by methods other than top-down decomposition. Although it is true that hierarchical abstraction is often used within an operational specification to defer details, these details must be resolved before the specification phase comes to an end."

- C. In motivating an object-oriented program design strategy, [BOOCH83a] states:

"No matter what the particular application, the problem space is rooted somewhere in the real world, and the solution space is implemented by a combination of software and hardware. ...in the problem space we have some real-world objects, each of which has a set of appropriate operations....

"Whenever we develop a software system, we either model a real-world problem entirely in software or, in the case of an embedded computer system, take real-world objects and transform them in software and hardware to produce real-world results. No matter what the implementation, our solution space parallels the problem space. ...the programmer abstracts the objects in the problem space and implements the abstraction in software. ...

"Intuitively, it is clear that the closer the solution space maps to our concept of the problem space, the better we can achieve our goals of modifiability, efficiency, reliability, and understandability. ... all things we know in the real world are abstractions, and, if our solutions are distant from the problem space, we must make a mental or physical transformation to the real-world abstractions, thus increasing the complexity of our solution."

We acknowledge that in transforming the prototype into a solution oriented design and implementation, partitioning of functionality among modules will change. However, if an object-oriented development methodology is used throughout, some of the user- or problem-oriented objects in the prototype may persist into the final system. Then, if both the prototype and the final system are coded in the same language, some objects from the prototype may be reused in the final system.

blank back page

SECTION 4

A MODEL OF COMPUTATION FOR PROTOTYPING

This section describes the model of computation needed to develop object-oriented prototypes and suggests one way of implementing this model. As background for presenting the computational model for prototyping, this section discusses some classical models. The ideas presented here result from work on Task H1.6 and from Honeywell funded work.

4.1 PROBLEM STATEMENT

Programs are specifications of computations. Computational specifications are meaningful only when interpreted in the context of a computational model. Attempts to specify a computation without understanding its underlying computational model usually result in errors caused by clashes between the semantics of the actual model and the semantics the specifier assumes. Computing with digital computers began with simple models for simple computations. As computations became more complex, more comprehensive models were developed. Prototyping is a specialized task that will require a specialized model of computation. The model for prototyping is driven by the need to modify portions of a prototype as it runs without affecting other portions of the prototype.

4.2 OUTCOME

We used the results of the RaPIER methodology and reusability work to determine the requirements for a model of computation for prototyping, and proposed a model of computation that appears to meet those requirements.

4.3 THREE CLASSICAL MODELS OF COMPUTATION

This subsection surveys three classical models of computation: the batch, time-sharing, and transaction processing models. Each has shortcomings for supporting prototyping. It was an analysis of these shortcomings for the new demands of prototyping that led to the type-manager model proposed in subsection 4.4.

4.3.1 Batch Processing

The batch-processing model is the oldest model of computation. There is one active agent in a batch job: a process acting on behalf of a single active program. This process is not connected to any other entity; that is, the process does not act on behalf of some user, or as part of some project or department, and it is not accountable for resources to some financial account.

In the middle 1960's, batch-processing systems began to process jobs for more than one user at time, and processes began to be associated with a user or account. However, this association was for bookkeeping reasons and rarely controlled the resources the user was allowed to have or the services the user was allowed to request. The batch model is not comprehensive enough for prototyping, since prototyping requires significant user-interaction not possible in a batch environment.

4.3.2 Time Sharing

Time-sharing is a method of using computers that allows many users and processes acting on behalf of those users to interactively share a system and its resources. When time-sharing became a general mode of computer use, we discovered that the batch model of computation was too limited. Specifically, users started to get in each other's way. When a process acting on behalf of a user could access all the computer's resources, and when several processes were active "simultaneously," limits were needed on what each process could do. The driving question in defining a time-sharing model of computation was: How are a computer's resources to be shared among many users whose activities are interleaved?

For time-sharing, an operating system executive schedules the use of the computer's resources among many users or user processes, each of which is identified by some authentication entity. Each system has its own entity definition scheme to use in authenticating resource-use by sets of users.(1) A time sharing operating system is also capable of managing a dialog between users and the system. The dialog takes the form of user commands and system activity in response. Time sharing operating systems also manage the connection of processes to programs and users. On most operating systems, the model

-
- (1) An authentication entity is an identifier set used for validating access to resources and services. Some of the more common views of authentication entities are:

VAX/VMS	Person.Group
G6M400	Person.Project.Type
OS/MVT	Person.Account
Multics	Person.Project.Type.Ring.AIMLevel.category-cross
ALS-APSE	Person.Project.Role.Tool
Unix	Person/Group & Tool(Person/Group)

is one login entity, one program. Some advanced time-sharing systems have a stack model of programs that can be suspended and resumed. A few systems have a model where one login entity can switch among many processes.

The time-sharing model, with its multiple processes, user interaction capabilities, and resource control is closer to the needs for prototyping than the batch model. However, the one user, one process, one interaction point model of most time-sharing systems is too limited for prototyping. This is true because, for example, a prototype builder may want to conduct simultaneous dialogs with a software repository, a PSDL processor and the prototype under construction. Although the builder will deal with only one dialog at a time, all three must remain open concurrently.

In order to manage shared resources efficiently, time-sharing systems were built with highly intertwined code and data. This intertwining makes it nearly impossible to modify portions of the system without affecting all users. For prototyping, one needs to modify parts of a prototype without affecting other parts of that prototype. Thus a computational model for prototyping must allow builders to construct prototypes from loosely coupled components, so that a prototype component can be modified without interfering with other parts of the prototype under construction or use.

4.3.3 Transaction Processing

In a transaction processing system a relatively large number of users use the system simultaneously. They make frequent event-demands (requests for transactions) that require relatively small amounts of computation on a small subset of the data potentially available to them. Many events take place during one user session with the system. The questions driving the transaction model of computation are: {1} How does one attach those users to the needed data efficiently and quickly? {2} How does one keep the state information associated with a user when (s)he is not attached? {3} Who is the user? What needs to be authenticated?

In both the batch and the timesharing models, a user is permanently attached to a process and the process acquires, on behalf of the user, fairly static resources that are validated at acquisition-time. This model of a session-long process-user connection is inappropriate in a transaction-processing environment because the cost of creating a process and attaching available resources is too high for short-lived processes, compared to the cost of queueing users to a free process. The alternate of having the required number of processes available at all times, whether in use or not, is not feasible because most operating systems used in transaction-processing are unable to keep the required numbers of processes in existence simultaneously. Even if processes were free, the time-sharing model would not solve the transaction-processing resource usage questions because the granularity of control provided by time-sharing operating systems is not fine enough (e.g., file rather than record/fields). Therefore in transaction processing, an event causes a user to be attached to a process and resources attached to that process based on what the process will be doing.

Both machines, for example a particular automated teller-machine, and individuals need authentication. Initial authentication and additional authentication for particular transactions raise a need to create and change authentication. Tying authentication to processes, as is done in time-sharing, is not a solution to resource-control in transaction-processing because users share processes serially. Instead, since not very much information about users or what they were doing needs to be saved between transactions, state is associated with the users' physical access line and referenced by the attached process when making authentication decisions.

To summarize, in transaction processing, processes and their system-mediated resources and services are relatively static while users and low-level needs are dynamic. The transaction-processing computational model, therefore, is one where there are static processes, but transient users and resource needs. A proper transaction processing model is thus one where users and their authentication are shared among processes. State-information appropriate both to what users are doing and to their current authentication is attached to a process for the duration of execution of a transaction. That state is then saved until the next need for computation, while the process is reassigned to other needs.

While transaction processing systems appear to provide a wide variety of displays and specialized user-input paradigms thus being a good platform for prototyping, the variety is illusory. The driving concern in transaction-processing is efficiency. There are tens and sometimes thousands of people at terminals. What is displayed, and how input data are processed, is largely "canned;" there is no room for modification without adversely affecting efficiency. In prototyping, the concern is modifiability; not efficiency. Thus the transaction processing model of computation will not support prototyping well.

4.4 THE CONCEPTUAL MODEL

This subsection discusses our requirements on a model of computation for prototyping and proposes a model.

4.4.1 Requirements for the Prototyping Model

We have already concluded that prototyping cannot be supported well by a batch, time-sharing or transaction-processing model. An acceptable computational model for prototyping must address the shortcomings of the classical models of computation in the face of new demands. Creation and use of prototypes place different requirements on the model of computation.

There are three separate interaction points between a user and a prototype in execution:

- o User interaction with the prototype itself;
- o User interaction with prototype's environment;
- o User interaction to change the prototype.

Each user interaction manipulates a portion of the total computation. In defining a prototyping model, we are concerned both with providing the user with a comfortable view for carrying out the three interactions cited, and with what a prototyping system can support. In addition, the prototype interacts with other programs that comprise its environment. That environment may also have to be modified to demonstrate alternatives in the prototype. Therefore, a model of computation for prototyping must support easy interaction with the prototype, modifying the prototype, and modifying the environment in which that prototype is executing.

RaPIER prototypes will be built from generic reusable software parts that are customized for a particular prototype by such means as parameters. These parts will be objects in the SMALLTALK sense. The prototyping model of computation must include the notions of object and of inter-object communication. The model's implementation must support both these notions.

Modifiability is the issue that "forces the model" in prototyping, as resource-sharing forced the model in time-sharing and transaction-events forced the model in transaction processing. Users must be able to modify a prototype system rapidly so that they can then observe alternatives. When a change is made in a prototype, the rest of the prototyping experiment must not try users' patience with too much repetition of things already seen. Thus, a prototyping system be structured so that:

- o prototypes can be built quickly;
- o already built prototypes can be rebuilt quickly;
- o rebuilding while a prototype execution is underway does not require restarting a prototype experiment from the beginning.

In contrast to most other types of systems, where efficiency is a prime driver and design choices are made that reduce the system's generality and modifiability, prototyping requires modifiability and generality in order that prototypes can be built and changed quickly. This need implies that most operating systems and system software, being designed with opposite objectives to those of prototyping, are not well suited for prototyping.

4.4.2 The Model

Systems that can be modified rapidly will have two characteristics in common: {1} they use well-defined objects; {2} those objects are "glued" together in a small number of well-defined ways with glues of appropriate binding-strength. Thus, a prototyping model of computation will have to define what objects are and how objects are connected/separated.

4.4.2.1 Objects

In order to be able to replace user-visible objects "at will" during prototype execution without causing havoc, those objects must be defined and connected in a special manner. Both code and data will have to be replacable during prototype execution almost without restriction. This "late binding" capability can be provided by a facility known as "dynamic linking." Languages such as Lisp support late binding/dynamic linking naturally. Some of Lisp's characteristics that provide this support are interpretation, code and data with similar representations, dynamic scoping, and user-transparent multiple call-targets. Others languages such as PL/I and Ada can be dynamic in appropriate operating systems environments such as Multics and R1000. Characteristics of other languages, such as FORTRAN, make change/replacement nearly impossible. However, late binding is not enough. One needs to be able to unbind and rebind both code and data where appropriate.

The organization we recommend for prototype programs, to maximize both reuse and modifiability, is that of a collection of typed objects, with type-managers responsible for all operations performed on the objects. There must also be minimal linkage between various types of objects; what linkage there is must require little or no knowledge by either the programmers or the run-time system of the types of objects being linked.

A type manager model does not necessarily imply the existence of a run-time type-manager that examines and manages all messages between objects, or the particular developmental style this forces. Type managers can be implemented in at least the following ways:

- o A program executes "send <message>" and an object is forced to do something specific to determine if it wants the message and to receive it.
- o A program executes "send <object> <message>", thus specifying what object picks up the message.
- o A program executes "send <method> <message>", thus specifying a set of objects that may pick up the message.
- o A program executes "send <type-manager> <object> <message>", thus selecting objects in a two-level name-space.
- o A program executes "call <type-manager> <object> <message>", thus achieving the same as above, but with much less run-time overhead.

Note that the last alternative is similar to the conventional call-return programming language style.

One of the key concepts in implementing any type-manager model is whether the number and kind of operations is frozen or evolutionary. By frozen, we mean that the number of operations each type manager manages is fixed, and that each operation's number and types of parameters is also fixed. By evolutionary, we mean the type-manager for a single object handles a varying number of operations, not all of which need to be specified at the same time, and that the number and types of parameters may vary. It is possible to have a fixed-set of operations for checkability plus a variable-set for extensibility.

A Model of Computation for Prototyping

Inheritance of properties and operations, such as is found in Lisp flavors and SMALLTALK objects, increases the speed with which prototyping software can be built and altered by allowing new objects to be built as specialized or combined forms of existing objects. Our prototyping model uses Ada's generic ability for its inheritance mechanism and Ada's overloading-resolution ability to make the proper linkage between routines.

We will assume that any object on which an operation is to be performed will either be operated upon, or an exception will be signalled if the operation cannot be performed. That is, we assume there are no resource authorization constraints. Objects or references to objects are passed to their type-managers, which perform the actual operations. Our assumption is justified because prototyping is performed in a single user or small cooperating group setting where there is no willful bad code. Thus only authorization to detect accidental errors is required.

Here are some open questions: Should objects be active (for example, tasks or processes) or passive (for example, subprograms)? If they are active, are they to interpret arbitrary messages, and handle them in similar ways to Lisp flavors, or should they handle only defined operation calls? What is the exact style of inheritance; how similar will it be to SMALLTALK's hierarchy or Lisp flavors' lattice?

4.4.2.2 Glue

For prototype development, the model must also allow the developer to glue together reusable parts rapidly. "Glues" include:

Paradigm	Examples	Remarks
binding-units	any-linker	
dynamic-linking	snobol,lisp,Multics	
command-files	Unix,VM/CMS	
parameterized-full-programs	Multics-EMACS & -Compose	
inheritance-from-parent	Unix-commands	
inheritance-from-component	Ada-with,lisp-flavors	
embedded-semantic-in-data	strings-as-pathnames	
trapped-instructions	missing-floating-point	
virtual-memory	Multics-fgbg	
common-files	GCOS8-talk	
input-output-streams	Multics-14E27A	
interposed-entities	Multics-cu_\$cp	
messages	Thoth-Godfather	
inter-process-signals	lisp-throw-catch	
intra-process-signals	Multics-cleanup	
expection-propogation	Ada-exceptions	
active-functions	Multics-underline	
generated-from-data	Ada-repository-menus	database contents
inter-language-calls	FORTTRAN-Numeric-constraints	
generic-instantiations	Ada-generalized-stack	
built-from-high-level-descriptions	RaPIER-PSDL	
macro-processed-special-form	SNOBOL-implementation	

It is an open question how many kinds of glue RaPIER will need to support, given the type of components we plan to use. It is also not clear how much glue needs to be visible and accessible in the prototype execution environment. Nor is it clear how tightly the construction and execution environments must be bound together.

4.5 IMPLEMENTING THE MODEL

The major notions in the computational model we propose for prototyping are reusable composable objects, type-managers and late-binding. This subsection discusses implementation support for these notions.

4.5.1 Objects

The Ada package will implement the object notion. RaPIER will contain a software base of reusable parts that are Ada packages. There may be packages in the software base that were designed and implemented outside of the RaPIER system, but those packages designed specifically for RaPIER and prototyping will be written using the reusability guidelines contained in section 6 of

this report. We expect object-oriented Ada packages written according to the reusability guidelines to be reasonably easy to find, understand and customize for use in a particular prototype. In addition, we expect their object-orientation to help ensure their separation from other factors in their environment of execution; this separation will enhance their modifiability.

The low-level building blocks from which prototypes are synthesized will have the following characteristics:

- o Minimum dependency (in the specification) on other packages.
- o Types, constants, and exceptions are externally visible.
- o No externally-visible data-objects.
- o Where appropriate, internal data-object manipulation synchronized (e.g., by tasks) to avoid inconsistent operations.
- o Manipulable data-objects will normally be created and destroyed by routines in the unit providing the service; thus every low-level package will have creation/destruction routines.
- o Low-level packages will be defined in a straightforward way without special optimizations for one attribute (e.g., storage) at the expense of another (e.g., time).
- o Reusable parts will be table-driven instead of logic-driven to enable their insertion in wide variety of prototypes.

Prototypes may also contain large, table-driven components (for example, a text-formatter) that will not be built from low-level units. They may be bound-units, that is object code, or be written a non-cooperating language. These large components will also be catalogued in the software base and will be incorporated into the prototype during the build process. However, we do not expect them to be catalogued in the same way as reusable Ada objects, since their incorporation at the object-code level rather than the source-code level will require different information from that used to incorporate Ada source objects.

4.5.2 Glue

Medium sized modules will be synthesized from lower-level components based on a PSDL specification of function and behavior. These units will be glued together using Ada's normal calling conventions. The main-program will also be synthesized by this method.

The large, table-driven units will require a different form of glue. These units will be glued by dynamic linking, calls through the command processor, and by parameter files of data.

4.6 FUTURE WORK

Different tasks require different models of computation. Prototyping requires a model of multi-task interaction, and one where developed components or parts of components can be quickly and easily replaced during prototype execution. The effects of that replacement must be minor, and each effect must be known at the time of the replacement.

The work needed to realize the proposed model of computation includes:

- o Complete definition of the model. This section mentions open questions about the nature of objects, about inheritance and about glue types. These questions must be answered before the conceptual model is complete and before a complete implementation of the model can be proposed.
- o PSDL to Ada mapping. We must investigate how to map PSDL to Ada. While the mapping appears to be straightforward when there is only one type of object, and objects do not interact, it is not clear how objects should be mapped when there are interactions, such as that between a database and a user-interface menu.
- o Component Generality. We must discover what level of generality is adequate for prototyping in a particular environment. While there is good data in some areas, for example, PL/I in a multi-user resource-sharing environment of screen-oriented character-terminals, we know of none for Ada in today's bit-mapped, distributed environment.
- o Run-Time Model. We must investigate what is the best run-time model for supporting type-managers in an Ada environment that requires dynamic changes to running prototypes.
- o Glue. We must characterize the glue types needed in RaPIER. We believe that using only one kind of glue is inadequate. We do not yet know the characteristics of our target environments well enough to predict which glues are right, either for initial prototype construction or for modification during use.

SECTION 5

SOFTWARE BASE - CLASSIFICATION SCHEME

This section describes initial work on software parts classification. The work consists of both the classification scheme and a discussion of use-scenarios involving a software base browser and a software base that supports our scheme. We have also suggested an implementation strategy. The work on the software base is intended to complement work done by International Software Systems, Inc. (ISSI) under separate contract with the Office of Naval Research and to enable us to specify Honeywell's requirements to ISSI.

5.1 PROBLEM STATEMENT

The ease of locating a potentially reusable software part is a major incentive for the user to want to search for and reuse such a part. Many of today's software repositories (for example the Ada SIMTEL repository on ARPANET) are not easy to use especially if the user is not familiar with the names of the software parts they contain and cannot guess portions of some of the keywords needed to retrieve the part. Modern database management systems especially relational databases allow the user to retrieve candidate software parts through the use of associative queries. This flexibility is provided by putting some values in database attributes (for example name-of-author, function, timestamp, or no-lines-of-code). This classification (flexible as it has become) is based on the syntax of values in database attributes rather than on semantics.

The classification of objects in modern object-oriented programming systems such as SMALLTALK [GOLDBERG83] is more meaningful than database classifications in the sense that the system has generic hierarchies of objects that have been built up based on semantics. But outside of the small set of predefined objects which the original implementors of the system have provided, it is not clear (as we shall explain later) that such small object-oriented systems can be used unerroneously for frequent insertions and deletions of software parts by multiple users. Our job has been to formalize a similar classification scheme for large, shared software bases.

5.2 OUTCOME

We started this task by studying some of the existing software repositories such as the Ada SIMTEL repository. As mentioned earlier, those repositories are organized by loosely classifying software according to functionality; retrieval is based on keywords, partial-string matches, and the perusal of listings of code and documentation. In a system for the rapid construction of software using prototypes, there is need for a conceptual basis for classifying software so as to guide the search and reduce retrieval time.

We also studied some of the new object-oriented systems such as the SMALLTALK system [GOLDBERG83] and the Symbolics [SYMBOLICS84] Flavor System. Those systems are well-organized but small and not very well oriented for use in a multi-user environment. We have pursued a similar classification scheme; this time, it is based on formal specifications rather than on the intuition of a single user.

We have also studied use-scenarios involving users and their interaction with a software base browser. The study enabled us to define functional characteristics of the software base browser as well as the interfaces that the software base must provide in order to support those functions.

An approach to implementing our classification scheme is included in the report. We note though that this part of the work was done in January 1986 using internal Honeywell funds.

5.3 INTRODUCTION

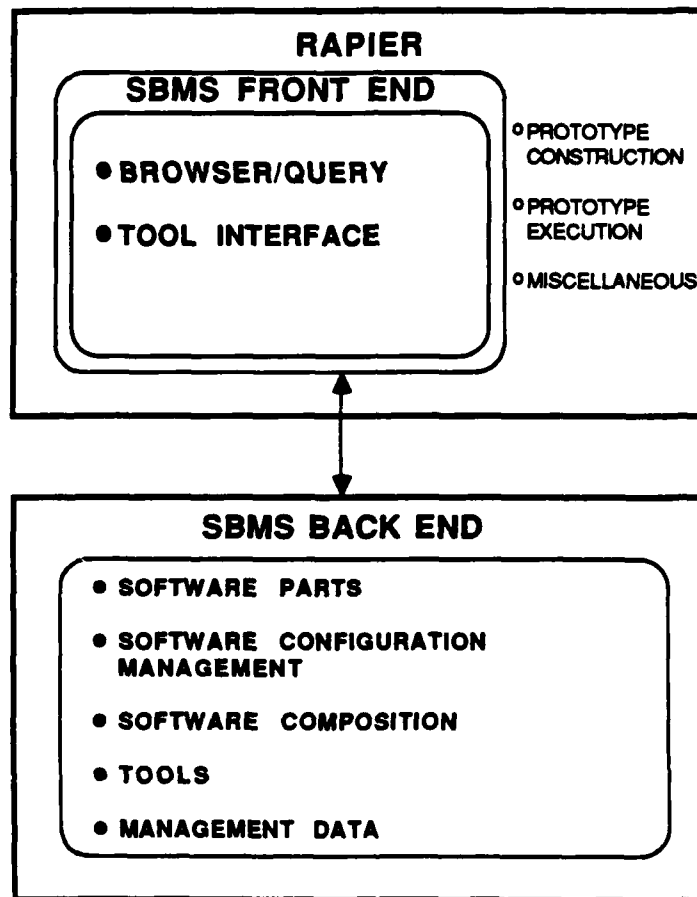
Database management systems (DBMS) are useful to the business data-processing community because they shield the user from the physical implementation of the database. A user's view of the data can correspond to his/her view of the real-world; it is then the duty of the DBMS to map such a conceptual view onto the physical implementation of the database. Similarly, a Software Base Management System (SBMS) should allow a developer to deal with conceptual objects irrespective of their underlying implementations. As with data in a database, the SBMS is responsible for mapping the developer's conceptual view of an object to some physical implementation of the object at the appropriate time.

We are interested in such an SBMS. Some of the major functional requirements of an SBMS (for example configuration control) are discussed in [ONUEGBE85a]. Another report [ONUEGBE85b] presents a methodology for classifying the software in an SBMS; we call this classification scheme Behavior Abstraction. Behavior Abstraction will work best in a fully-automated software library where software is functionally specified and where such specifications can be evaluated by some engine.

Since the technology for processing functional specifications has yet to mature, we adopt a phased approach to implementing Behavior Abstraction.

Software Base - Classification Scheme

Phase 1 consists of an SBMS that supports a semantic data model. The external schema resides on a front-end while the SBMS is hosted on a back-end as in Figure 5-1. The classification of the software parts is based on formal functional specifications but it is a manual process at this stage. A browser, a query language, and a high-level language for specifying the behavior of prototypes all serve to interrogate the SBMS and to retrieve the software parts for reuse.



File No. 6-0368

Figure 5-1 Initial RaPIER Functional Architecture

Phase 2 includes all the features of Phase 1 along with automatic update of the structural relationships in the SBMS whenever a software part is added or deleted. This phase involves interfacing the SBMS to syntax-directed editors, compilers, and "parser-like" entry tools.

Phase 3 includes all of Phase 2 along with Behavior Abstraction. Software classification at this stage is based on machine-processable formal specifications and is automatic. The user's interface to the SBMS is a catalogue of abstracted behaviors, conceptual objects; the binding of implementation to behavior is delayed. The following benefits accrue from such a scheme:

- o because of later binding of behavior to code, we achieve the kind of flexibility that makes it easier to reuse designs/specifications.
- o we support an object-oriented construction methodology at the front-end; the user gets comfortable with a stable set of objects in the catalogue rather than with the many constantly changing implementation names.

5.4 THE BEHAVIOR ABSTRACTION CLASSIFICATION SCHEME

A classification scheme that models user perceptions of module functions is critical to the success of the SBMS. Previous classification schemes stress loose groupings of software parts by their functions. For example, all mathematical functions could belong to one group while logic functions get grouped separately [EDWARDS77]. Keywords and partial string searches have been helpful in retrieving modules so long as the user knows or can guess some portion of the module's name.

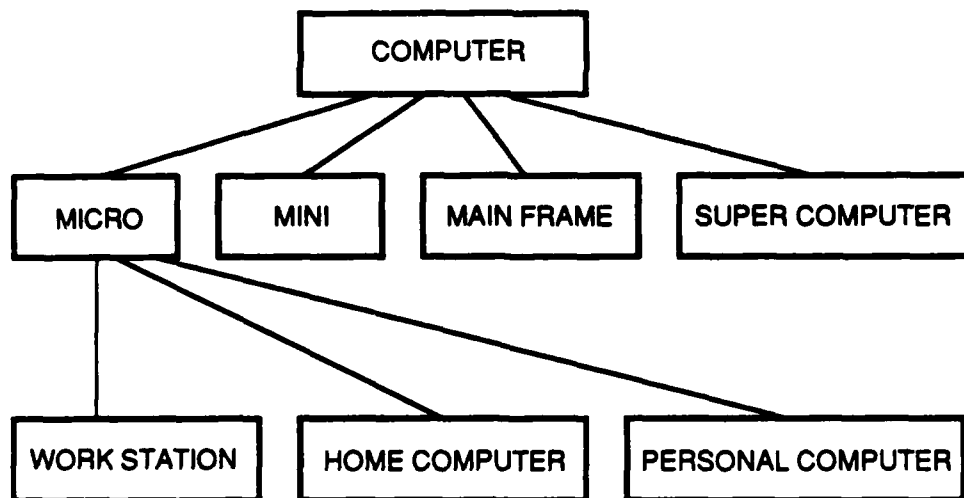
The use of modern database management systems, especially relational databases, enables a user to retrieve a class of software components through associative queries [YEH84]. For example a user can retrieve all mathematical functions that were written by a named author before a specified date so long as those attributes exist in a database. The names of the modules that meet the query specifications are then used during software construction.

Even though relational systems are a major improvement over the older cataloguing schemes, the structural properties as well as the semantics of software modules are hard to describe using relations. For example, if the category of trigonometric functions is a subcategory of mathematical functions, there is no easy way to capture this in a relational database. Improvements in the modelling power of databases, especially the semantic models as well as advances in object-oriented programming, provide the impetus for more powerful classification methodologies. We will discuss these below.

5.4.1 Semantic Modelling and Object-oriented Classification

Semantic modelling grew out of the programming language, database, and artificial intelligence communities in order to improve the semantic expressiveness of databases and other information repositories. We will not dwell so much on the expressive power of the semantic database models since these are discussed in [HAMMER81, BRODIE81]. We will, however, discuss two modelling concepts that are useful in the SBMS.

- i. Generalization Hierarchies. A generalization hierarchy is one in which classes and sub-classes are defined. A class is defined by a set of attributes; all objects possessing those attributes are grouped together. A sub-class is a specialization within a class. This age-old notion of taxonomies was incorporated into data modelling in [SMITH77]. Figure 5-2 illustrates a generalization hierarchy for computers. MICRO, MINI, MAINFRAME, SUPERCOMPUTER, are all sub-classes within the class of COMPUTER. Similarly, WORKSTATION, HOME COMPUTER, and PERSONAL COMPUTER are all subclasses within the class of MICRO. In a similar fashion, a behavioral hierarchy for software parts can be modelled in an SBMS that directly supports generic hierarchies.



File No. 6-0369

Figure 5-2 A Generic Hierarchy of Computers

- ii. Composition Graphs. A software part may be a composite that references or is made up of other software parts, which in turn may reference or include other software parts, possibly recursively. Even though existing databases (for example relational or network) can be used to capture the composite nature of a software part, they do not have a direct way to represent a composite object. For example, in the Symbolics Flavor System [SYMBOLICS84], a flavor is an object type and each flavor has a name and a set of methods (a method is an implementation of the response to the set of messages that an object understands). Bottom-up object-oriented programming using the flavor system begins with a collection of reusable flavors (objects); new flavors are built by combining more primitive ones. Eventually, one obtains the appropriate objects to solve the problem at hand. The Flavor Examiner on the Symbolics provides the means by which the user traverses this composition graph in order to find potentially reusable objects.

5.4.2 Building Prototypes With Reusable Objects

The concept of an object as a named computational entity with an identifiable behavior is central to object-oriented programming. An object's behavior is its reaction to the set of messages it "understands," where a message is a request to initiate processing or provide information, and "understanding" means possessing a defined response. Messages are akin to conventional procedure calls with the distinction that the sender has no idea how the receiver implements its request. Inside the receiver, messages are handled by directing further messages to other objects, and or by performing activities [RENTCH82]. Procedure calls are commands to carry out specific algorithms, while messages are requests to accomplish some activity by whatever algorithm is appropriate, where the object receiving the message decides what is appropriate. Similar objects (i.e. objects exhibiting some of the same behavior) constitute a class. The same abstraction mechanisms discussed for databases are used to organize objects into a class hierarchy.

We illustrate this using the Flavor System on the Symbolics 3600 LISP machine. A flavor is essentially an object type; every flavor has a name and a set of methods. A method is an implementation of the response to one of the messages an object understands. A flavor is similar to an Ada generic unit in many respects. An instantiation of a flavor receives requests for services called messages and may also respond to those messages. A message has a name and appropriate parameters.

Object-oriented programming can be carried out top-down or bottom-up. Top-down object-oriented programming comprises six activities[BOOCH83]:

1. Define an informal strategy for solving the problem at hand,
2. Identify the objects (nouns) in the informal strategy,
3. Identify each objects operations (verbs) in the informal strategy,

4. Define each object's interface: -- the services and information it offers to other objects,
5. Implement each of the objects
6. Implement the informal strategy as a program that uses these objects

In this case, an implementation is developed for each object needed.

Bottom-up object-oriented programming begins with a collection of reusable software objects such as the SMALLTALK system objects, the collection of flavors on the Symbolics, or a user's personal library. These objects are well-documented, are usually reliable, and the system is usually well-organized. Objects for the problem at hand are built up by combining more primitive (system or user-defined) objects. Eventually, the system contains the appropriate objects to solve the problem at hand. Bottom-up object oriented-programming is a natural way to exploit a software repository's resources.

The situation with today's commercially available object-oriented computing systems, however, is that they often contain well-known predefined objects and objects which a single user, or a small group of users have added. The RaPIER situation is different. First, the SBMS must contain a large number of software parts contributed by a large number of users (so long as the software is written according to the reusability guidelines suggested in section 6). Secondly, unlike in today's systems where there is often only one software part per object type or class, an SBMS class may be populated with numerous software parts all exhibiting the same behavior. A useful SBMS in such a situation is one in which software parts can be automatically classified according to some semantics. Also, the flexibility and speed needed for building prototypes requires that behaviors rather than software part names be incorporated into the Prototype System Definition Language, PSDL. This way, designs are insulated from the insertions/deletions in the SBMS. The binding of a software part to the behavior it implements takes place as late as possible. When once the first prototype is built and its behavior has been exercised, further tuning may then occur and the user can then begin selecting among software parts by including further restrictions in the PSDL.

5.4.3 The Classification Scheme in RaPIER

The Behavior Abstraction classification scheme [ONUEGBE85b] builds generic hierarchies out of a set of functional specifications. This is similar to the SMALLTALK object hierarchy. In the RaPIER however, each behavior class is a formally specified abstract object. For each such abstract object, there can be more than one alternative implementations. This is similar to the notion of versions and alternatives in a design database[KATZ84]. Discriminants are then used to restrict retrieval whenever the user needs one of those implementations. The task of identifying which implementation belongs to which abstract object is the task of a theorem prover or the human librarian.

In the next subsection, we will discuss the various phases of implementation of the Behavior Abstraction classification. The following terms used in this paper are now defined in the context of RaPIER and Behavior Abstraction.

Definition 1. A catalogue is the set of behaviors that can be attributed to all of the components in the software base. In RaPIER's SBMS, a catalogue consists of a set of generic behavior hierarchies.

Definition 2. A behavior is the state change caused by an object's response to one of the requests it understands.

Definition 3. An object is a set of related messages. Each object type is unique - for example "table manager," "window," or "mouse". In the SBMS, software parts which exhibit the same behavior are grouped as a class.

Definition 4. A message is a pair <object, request> where object is the name of an object and request is the name of one of the object's behaviors.

Definition 5. Behavior Abstraction is the functional specification of objects and their arrangement into generic behavior hierarchies in SBMS.

Definition 6. Delayed binding is the binding of an implementation of an object to an abstract message <object, request>. Delayed binding is similar to the retrieval of data in response to a query. When Behavior Abstraction is fully implemented, abstract messages rather than implementation names will be used in PSDL.

5.5 THE PHASED IMPLEMENTATION OF BEHAVIOR ABSTRACTION

5.5.1 PHASE 1 - MANUAL CLASSIFICATION

Figure 5-3 illustrates a possible architecture of RaPIER's SBMS during Phase 1 of the implementation. The back-end consists of a software base implemented in an extended E-R model (for example the Entity-Category-Relationship, E-C-R model developed at Honeywell extends the E-R model to include the notion of generalization abstractions [WEELDREYER80]). The front-end consists of a software base browser, query processor, and PSDL. Reusable software parts are identified through browsing and querying; the names of reusable parts are then incorporated into PSDL. We will discuss the major portion of the schema for implementing the classification scheme in SBMS, the equivalent external schema as seen by the user in the front-end, the software base browser, and the manual classification of the software parts. The PSDL component is being defined and built by an independent subcontractor(1) and will not be described in this paper.

(1) For information on PSDL, please contact Prof. R. Yeh, International Software Systems Inc., 12710 Research Blvd., Suite 301, Austin, TX 78759.

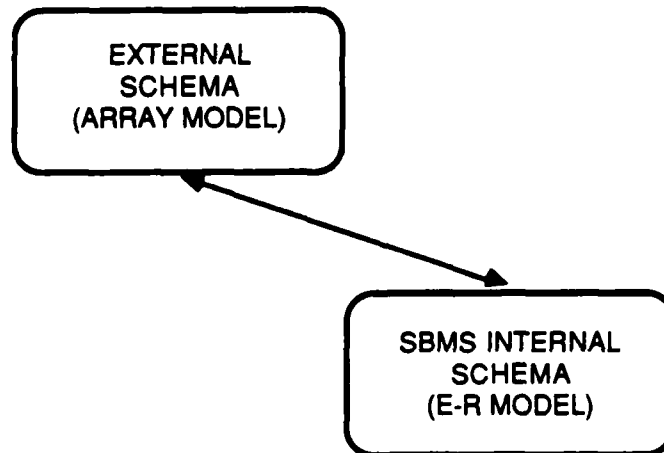
5.5.1.1 The SBMS Schema

The schema of SBMS is a very simple one. The entity type BEHAVIOR CATEGORY which along with the relationship IS A SUBCATEGORY serve to represent the generic hierarchy in the system. BEHAVIOR CATEGORY has a one-to-many relationship with the entity type IMPLEMENTATION which represents the implementation choices for each object class. IMPLEMENTATION entity type along with the relationship USES constitute a composition graph. The entity type INTERFACE represents the interfaces imported or exported by the various modules. Its attributes include names of importers and exporters and the input and output parameters among others. These structures (for representing the generic functional hierarchy and the code composition graph) are the most important ones in SBMS.

There are of course other entities such as those required to represent the data structures, but they are not very important for the purposes of this discussion.

The external schema is modeled using concepts from the Array Theoretic Model [MORE81, ONUEGBE85a] instead of the ECR model. This is done in order to support browsing. The Array model is a hierarchical data model in which objects are represented as nested arrays. It is therefore easier to present information in progressively greater detail as the user navigates the software base.

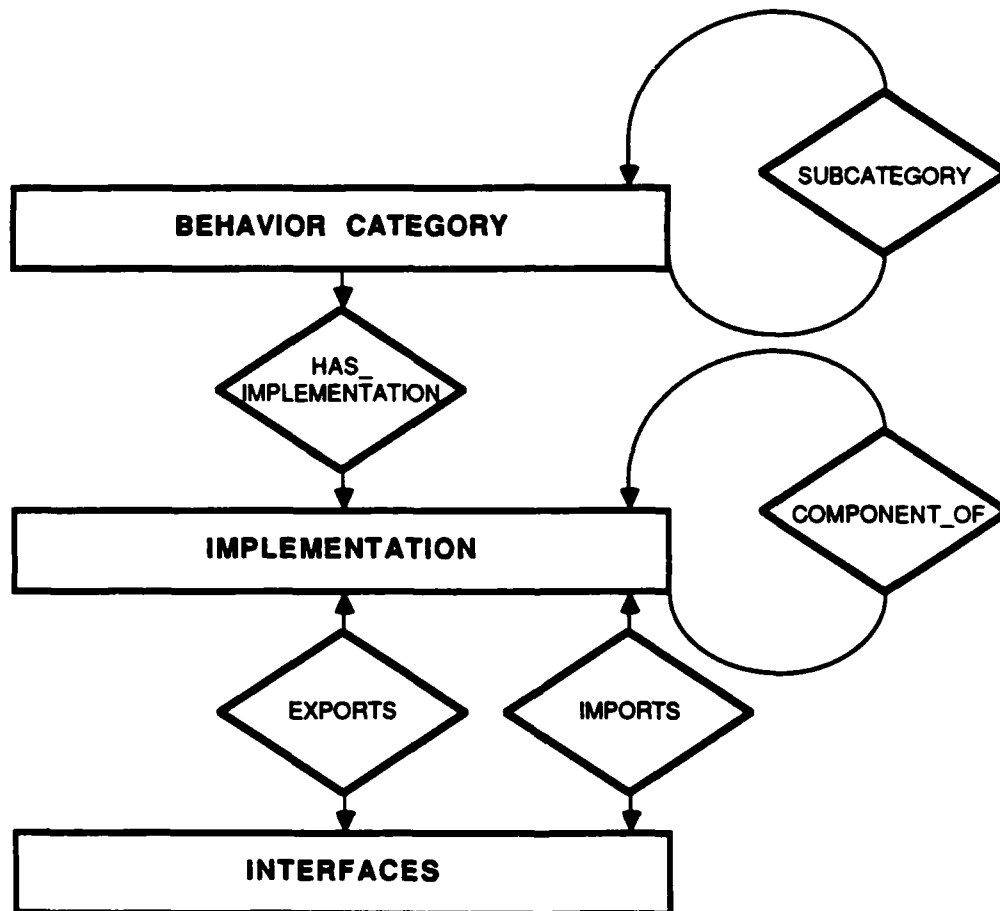
The mapping from ECR to the Array model is straightforward. To see how this works, we examine the schema of Figure 5-4 and 5-5.



File No. 6-0370

Figure 5-3 SBMS Functional Architecture for Phase 1

The object **BEHAVIOR CATEGORY** is the major object in the database. It has a unique system identifier, a name, the author's name, a timestamp (to say when the object was defined and approved) and a toolstamp (to say what tool was used in producing the object). The following arrays are needed to help the user navigate the generic behavior hierarchy (GBH) and are self explanatory: immediate subcategory, immediate supercategory, all subcategory and all supercategory. The array "functional_specs" contains the formal specification of the abstract object.



File No. 6-0371

Figure 5-4 E-R Schema for SBMS Software Parts

Software Base - Classification Scheme

BEHAVIOR CATEGORY:

```
id :
author :
timestamp :
immediate subcategory :
immediate supercategory :
all subcategory :
all supercategory :
functional specs :      /* functional specification for the abstract object */
implementation :
-
-
```

IMPLEMENTATION

```
id :
name :
type : (PACKAGE | SUBPROGRAM | TASK)
tool stamp :
immediate component :
immediate dependent :
all component :
all dependent :
discriminants (if package) :
version :
timestamp :
author :
imported interface :      /* implementation specific specifications */
exported interface :
-
-
```

INTERFACE (especially package specs)

```
id :
name :
specification :      /* implementation specific */
immediate exporter :
immediate importer :
all exporter :
all importer :
timestamp :
toolstamp :
version :
-
-
```

Figure 5-5 Array Model Version of Fig 5-4

The behaviors of software parts will be formalised. The array IMPLEMENTATION is the list of all implementations of an abstract object. The attributes of IMPLEMENTATION include id, name, toolstamp, version number, author, and other arrays which help the user navigate the composition graph (immediate component, immediate dependent, and so forth). The discriminants consist of those extra specifications that make the implementation different from the generic functional specs. Such discriminants include specifications concerning the environment, performance, reliability, and so forth. In Phase 1, the discriminants will be also be captured as database values (in order to support queries); eventually, however, they may be formally specified.

The implementations that will be classified will be Ada packages written according to the guidelines in section 6. Users who want to add tasks, procedures, and functions into the library must package them in order to make them visible. Users can, however, browse through other modules that are components of packages.

5.5.1.2 Browsing

Browsing in SBMS consists of navigation and probing [MOTR084] through the generic functional hierarchy and the composition graph. As soon as the browser is invoked for use in RaPIER, three major contexts are established as shown in Figure 5-6a. The first is the context for BEHAVIOR CATEGORY, the second is the context for IMPLEMENTATION, and the third is for INTERFACE. As a user browses through the hierarchy of functional specifications, data is displayed in a window as shown in Figure 5-6b.

In order to display data concerning a particular implementation, the user would have to pick on the name of a particular implementation in the BEHAVIOR CATEGORY context. In order to display information concerning a particular interface, the user has to pick on one from the IMPLEMENTATION context.

The windows described in Figure 5-6a and 5-6b are used for illustrating the concepts only. In practice, browsing in RaPIER is within the larger context of the Construction Environment, so the windows do not appear exactly as shown in the figures.

FUNCTIONAL CATEGORY:
IMPLEMENTATIONS:
INTERFACES:
COMMANDS:

Figure 5-6a SBMS Query/Browse Display Windows

FUNCTIONAL CATEGORY: id : 001 : : specs : "table manager" implementation : hash, tree, ... : :
IMPLEMENTATION: id : 0011 name : "binary_tree_table" type : PACKAGE tool stamp : waterloo C : :
INTERFACES: : :
COMMAND:

File No. 6-0372

Figure 5-6b SBMS Display - an Example

5.5.1.3 Classification

We will concentrate on building domain-specific libraries in order to simplify the initial RaPIER and to help us gain experience. Classification in RaPIER is a bottom-up process; a human librarian reads the functional specifications of the various software parts and decides which generic object classes should be created. It is also then librarian's responsibility to group implementations of an abstract object together. The librarian also builds a generic hierarchy of the abstract objects. The rules for the classification will be given later in this subsection. Also, for the composition graph, the librarian would have to read the implementations in order to identify the call or "with" chain of the Ada code. (This is a tedious process!)

The following rules guide the classification.

Rule 1: A behavior category B is said to be a subcategory of another category A, iff A inherits some or all of B's messages. For RaPIER, Inheritance applies only to the abstract objects and is used mainly for classification; since this is not an attempt to build a programming environment into SBMS, the concrete inheritance by the various implementations only helps the user of the browser to traverse the Composition Graph discussed earlier and has no other implications. The hierarchy of abstract objects constitutes a partial ordering of behaviors and the discipline of Behavior Abstraction is the construction of such partial orders.

Rule 2: An implementation is said to belong to a behavior category (abstract object) iff there exists a homomorphic mapping between the implementation's specification and that of the abstract object. A theorem prover can be used to enforce this [GUTTAG78], otherwise the human librarian would have to make that judgement. For example, if one implementation of a stack uses an array and another uses a list implementation, the theorem prover would have to establish the homomorphic mapping between each of these implementations and the generic object "stack" in order to classify those implementations as objects of type "stack." A future paper will present the two rules stated here in greater detail and present a classification of some embedded-system software using those rules.

To summarize, browsing in SBMS consists of navigation and probing of a generic functional hierarchy and a composition graph structure created by the dependencies among software parts. In Phase 1 of the implementation of Behavior Abstraction, a human librarian loads the database manually.

5.5.2 Phase 2 - Semi-Automatic Updates

We noted earlier that the composition graph of the SBMS is very tedious to update. Phase 2 implementation consists of providing interfacing the SBMS with an Ada syntax-directed editor interface so that the relationship HAS_COMPONENTS can be automatically updated. Also, a stand alone tool which uses the symbol table of a compilation to update the database should be available after Phase 2.

The choice of the appropriate node in the functional hierarchy for classifying the part is still a manual process though.

5.6 PHASE 3 - BEHAVIOR ABSTRACTION

This is the full automation phase. Both the functional hierarchy and the composition graph can be automatically updated and the correctness of a classification is enforced using a theorem prover. (The use of a theorem prover to help in the specification and manipulation of abstract objects in databases is being studied at this time [KEMPLE86]) Because the functional hierarchy is almost fully installed at this stage, there is a basis for inserting, deleting and retrieving functional specifications.

Browsing becomes more interesting at this stage since we can now browse using the semantics rather than the syntax of the software. Examples of interesting new primitives for the browsing include:

1. `get_implementation(specification)`. This serves to retrieve any implementation that has the behavior specified. The construct can be further restricted with "any" or "first" or "all."
2. `probe(partial_specs)`. Just as with partial string match of keywords, this construct retrieves all (for queries or browsing) or a first implementation that is similar in some way yet to be defined to the behavior specified.

We note that Behavior Abstraction is almost without any risk in RaPIER since the prototypes are disposable. This makes RaPIER a good research vehicle for such a concept.

5.7 INTERFACES

This subsection discusses various other features needed for browsing and suggests a programmatic interface to support the browsing tool. This work was completed earlier; it assumes that ISSI's SBMS is being built on top of an advanced relational database management system [ROUSSOPOULOS85]. It is possible that ISSI's underlying database has changed, but the underlying browsing concepts are still the same. The low-level procedures needed to support browsing are included in an appendix at the end of this section.

5.7.1 Introduction

The first kind of interface, the browser, enables a user who has little or no knowledge of the data or its logical organization to quickly "shop around" in order to identify objects of interest. The browser must display information concerning the logical organization of data as well as the data itself. The information about the logical organization enables the user to discover the

structure of the software base and to follow promising trails. The browser must allow the user to quickly update the attributes of objects, in the same way as a screen-oriented text-editor allows a user to edit text files and receive immediate results. The browser must be forgiving of a user's forgetfulness or inability to express a request precisely. For instance, the user should be able to modify and reexecute queries. Finally, the browser must respond in non-trivial ways to a user's request. For example, if a user asks for a compiler and cannot find one, the browser should tell the user about the compiler generators in the software base.

Another interface, the query language, allows the user to interrogate the software base in an ad-hoc non-procedural fashion. This document does not discuss the query language since it has already been defined as SQL-like by International Software Systems, Inc. (ISSI) of Austin, Texas which is responsible for developing the software base.

The programmatic interface is a set of interface procedures for accessing the software base from a user's program. The RaPIER front-end being developed by Honeywell will be hosted on a workstation which supports Lisp and Ada. So the software base interface procedures will be called remotely from LISP and Ada programs. We assume problem-free communication between the front-end and the software base.

This document is not a specification of the user-interface requirements for ISSI's software base. It merely states RaPIER's needs and so complements ISSI's own requirements. At the time of writing, we assume that ISSI's software base is an enhanced relational DBMS and that the reader of this document is familiar with such terms as relations, tuples, attributes, cardinality, and so forth. The next two subsections discuss the browser and programmatic interface requirements. The requirements are specified even more tersely using Honeywell's WELLMADE [BOYD78] language in the appendix to this section.

5.7.2 The Browser

The browser is a program that allows the user to have an interactive session with the database. We assume that the user is not familiar with the contents of the database, so the browser must be easy to use and thus allow the user to quickly understand the structure of the database and to follow promising trails. We assume that the browser can display or receive data from a screen-oriented graphics terminal/workstation. We also assume that screen management routines for scrolling the display up/down, for windowing, for graphics, and for editing input data are available to the browser software.

5.7.2.1 Features Needed for Browsing

In order for a naive user to browse quickly through or to update the database, the browser must have features that are analogous to those of a screen editor such as EMACS [STALLMAN81]. We will suggest those features without necessarily suggesting an implementation approach.

1. Editing Capability. This allows the user to directly modify data that is displayed on the screen much in the same way as one modifies text in an editing buffer. Crisp commands to enable the user to save the buffer, undo previous updates, or apply a specified modification to more than one tuple of a relation should be provided. We recommend that the update rights be reserved for the librarian (or the group of individuals performing that function) in order to avoid chaotic updates.

Scrolling of the data on the screen will also enable the user to "hop" around until information that is of interest is found. Scrolling is up or down, a tuple at a time or a screen-full of tuples at a time. [STONEBRAKER84] discusses an implementation approach that would make this possible. Some of the editing capability discussed here are already available in some of today's form-based query interfaces.

2. Query Modifications. This feature enables the user to recall a previous query, modify such query, and reexecute it. For instance, if a user issued a query, "FIND ALL MODULES WRITTEN BEFORE 1980", and modified it to, "FIND ALL MODULES WRITTEN BEFORE 1975", the immediate effect would be to probably lessen the number of module names in the buffer. If the user changes 1975 to 1982, then the number of tuples could be larger than for the two earlier forms of the query.
3. Quick Navigation. With relational query languages navigation is achieved by issuing join queries. Crisp ways of navigating should be provided to enable the user to follow relationship chains. For example, in a hierarchical schema, the user should have quick ways of going from parent node to children nodes and vice versa.
4. Context Switching. Two kinds of context switching are useful. The first kind is analogous to the use of buffers in a text editor. It should be possible to issue more than one query, direct the results of each query to a different buffer and manipulate the buffers as needed.
5. Probing. Sometimes, the user has only partial knowledge of the needed object; if the database has deductive reasoning capability, it is still possible to satisfy such partially formulated requests. The least requirement we place on the software base in this regard is to do partial-string matching against keywords.

To some extent, some of the features discussed here have already been implemented in one form or another by some researchers or computer software vendors. For example, the forms facilities that are offered with some DBMSs have some features of a browser. The Flavor Examiner facility on the Symbolics 3600 family of computers is a browser of a sort.

5.7.3 The Programmatic Interface

The programmatic interface provides the means of communicating the results of a retrieval request from the DBMSs buffers into the user's program buffers. It also provides the means of communicating update data from the user's program back to the DBMS.

A user's programs submits retrieval requests in the form of a pre-compiled or raw query; the DBMS processes the query and stores the results in its buffers. At this point, the transaction becomes a so-called conversational transaction. The user's program submits a buffer variable to be bound to the DBMS's buffer. Assuming the DBMS sets the cursor to point to the first tuple, the user's program should then be able to fetch the first tuple, fetch the next n tuples up or down from the cursor, move the cursor to a certain position in the buffer, restrict or enlarge the number of tuples in a buffer by issuing a new predicate, update tuples in the buffer, ask the DBMS to store back the contents of the buffer, and open or close the buffer. An example of this kind of communication is that between an EMACS buffer and an EMACS window.

Most of these features are included in the appendix as well as in [ROUSSOPOULOS85]. [STONEBRAKER84] suggests that line identifiers be used to identify tuples in the DBMS buffer. We suggest that tuple numbers be used instead because we are not sure that there will be a one-to-one correspondence between a line and a tuple. This way instead of issuing a request such as "FETCH Line #20," the user's program issues "FETCH Tuple #20." It is up to the DBMS to order the tuples in its buffers[LYNN82].

5.8 FUTURE WORK

The first RaPIER prototype should include the Phase 1 implementation of the classification scheme. To this end, we expect to demonstrate some of the concepts by the end of 1986. This demonstration will include the browser. We will evaluate the prototype and incorporate any desirable changes to the scheme. Phase 2 implementation will be pursued if funds are available. Phase 3 implementation will depend on the results of further research. An automated Rapid Prototyping system such as RaPIER should, however, evolve toward an automated classification scheme.

5.9 APPENDIX

(1) TYPE DEFINITIONS

package SB_INTERFACE_TYPE_DEFINITION is

-- This package defines constants and types used
 -- in main program and other packages.

```

type RELATION_ID      is LONG_INTEGER;
type TUPLE_ID         is LONG_INTEGER;
type SB_PARSED_QUERY_TYPE is STRING;
type SB_RESPONSE_TYPE is STRING;
type SB_QUERY_TYPE     is STRING;
type ATT_TYPE          is STRING;
type ADA_CODE          is STRING;
type SB_SPECIFICATION_TYPE is STRING;
type RESULT_TYPE       is (FAILURE , SUCCESS);
type SB_CLASSIFICATION_TYPE is STRING;
type SB_NODE_TYPE       is STRING;
type SB_CONTEXT_TYPE    is STRING;
type SB_HELP_REQUEST_TYPE is SB_QUERY_TYPE;
type SB_PREDICATE       is STRING;
type SB_CONTEXT_TYPE    is
    RECORD
        classification_name : SB_CLASSIFICATION_TYPE;
        node_type : SB_NODE_TYPE;
        tup_id : TUPLE_ID;
    end;
end;
```

(11) BROWSER

```
procedure previous_context (
    in current_node : (
        SB_NODE_TYPE | SB_RELATION_NAME ) ;
    out SB_context : SB_CONTEXT_TYPE );
```

Switch to a previous context.

```
procedure query_eval ( in SB_query : SB_QUERY_TYPE;
    out SB_response : SB_RESPONSE_TYPE );
```

Evaluate the query and return the response. The browser should respond in a non-trivial fashion to a user query. For example, if a user wants to browse through a piece of code that performs quick_sort and if the browser does not find such a piece of code, the browser should inform the user about other modules that perform similar functions-- there could be bubble sort packages. As another example, a user may request to use a non-existent compiler; if the compiler does not exist, the browser should inform the user about any existing compiler generator in case the user wants to generate the compiler in question. In short, there should be (at least) some "nearest neighbor" query capability and some elementary inferencing as a first cut.

```
procedure list_help ( in SB_help_request : SB_HELP_REQ_TYPE;
    SB_context : SB_CONTEXT_TYPE;
    out SB_help : STRING );
```

List the requested help information.

```
procedure restrict_previous_query
( in previous_query : SB_QUERY_TYPE ;
    further_restrictions : SB_PREDICATE;
    out SB_response : SB_RESPONSE_TYPE );
```

Add further restrictions to a previous query. This should reduce the number of tuples returned in response to the previous query.

```
procedure add_don't_cares (in previous_query : SB_QUERY_TYPE ;
    new_don't_cares : SB_PREDICATE;
    out SB_response : SB_RESPONSE_TYPE );
```

Add more don't cares to a previous query in order to enlarge the set of tuples returned in response to the previous query. This is the opposite of procedure restrict_previous_query().

Software Base - Classification Scheme

```
procedure browse_code ( in code_name : STRING;
                        out code : ADA_CODE );
```

Put the named piece of code under the control of the browser; a user may then inspect the code and use attributes of interest for further browsing.

```
procedure list_classification_node ( in module_name : STRING;
                                     in name_of_classification_scheme : SB_CLASSIFICATION_TYPE;
                                     out node_name : SB_CLASSIFICATION_TYPE_NODES );
```

Print the node_type of a classification scheme of which the named module is a member.

```
procedure list_super_types ( in (node_type | module_name) :
                             (SB_CLASSIFICATION_TYPE_NODE | STRING);
                             out node_names : array of (SB_CLASSIFICATION_TYPE_NODE) );
```

Given the either the name of a module or the name of the classification node type, return the names of the supertypes (nodes) for that node type or module. This allows the user to go up a classification tree or lattice.

```
procedure list_tree_supertypes ( in (node_name | module_name)
                                : (SB_CLASSIFICATION_NODE_TYPE | STRING);
                                out node_names : array of (SB_CLASSIFICATION_NODE_TYPE) );
```

List the classification hierarchy from the topmost nodes up to the specified node or the node of which the specified module is a member.

```
procedure list_sub_types ( in (node_type | module_name) :
                           (SB_CLASSIFICATION_TYPE_NODE | STRING);
                           out node_names : array of (SB_CLASSIFICATION_TYPE_NODE) );
```

Given either the name of a module or the name of the classification node type, return the names of the subtypes (nodes) for that node type or module. This allows the user to go up a classification tree or lattice.

```
procedure list_tree_subtypes ( in (node_name | modul_name) :
                               (SB_CLASSIFICATION_NODE_TYPE | STRING);
                               out node_names : array of (SB_CLASSIFICATION_NODE_TYPE) );
```

List the hierarchy of node types; the root starts from the given node or from the node type of which the given module is a member.

```
procedure list_relationships ( in (node_name | module_name)
                               : (SB_CLASSIFICATION_NODE_TYPE | STRING);
                               out relationship_descriptions : SB_NODE_RELATIONSHIPS; );
```

List the relationship between the specified node in a classification scheme (or the node to which the given module belongs) and the adjacent nodes.

```
procedure scroll(in direction : BOOLEAN;  
               n : INTEGER;  
               ) returns (SUCCESS | FAILURE);
```

Scroll the results of a query n tuples up or down the screen. The variable "direction" is a boolean that indicates whether to go up or down.

```
procedure elaborate(in keyval : (INTEGER | STRING | FLOAT | ...  
                                ;  
                                rel : REL_ID;  
                                tup : TUPLE_ID;  
                                )  
returns (SUCCESS | FAILURE );
```

Display the remaining attributes of the object whose key is being presented.

(iii) PROGRAMMATIC INTERFACE

```
procedure SB_close_relation(in relation_id : RELATION_IDS);
    returns (SUCCESS | FAILURE);
```

There will be no more requests for the tuples of this relation. The SB may then flush its buffer of this relation if necessary. The user of the SB does not have to make this call since it merely a courtesy call to help the SB manage its buffers.

```
procedure SB_execute(in SB_tokens : SB_PARSED_QUERY_TYPE;
                    out SB_response : SB_RESPONSE_TYPE
                    )
    returns (SUCCESS | FAILURE);
```

Execute an SBMS pre-compiled query and return a response. Every response consists of at least the following : a Boolean flag to indicate success or failure, error messages if the processing failed, the kind of data returned (e.g ADA source code or database values), number of tuples if database values, byte length of the code if code.

```
procedure SB_flush_tuples(in relation_id : RELATION_IDS)
    returns (SUCCESS | FAILURE);
```

Flush the SB buffers of the tuples of this relation. This again is a courtesy call.

```
procedure SB_get_first_tuple(in relation_id : RELATION_IDS;
                             out SB_response :SB_RESPONSE_TYPE)
    returns (NULL | FOUND) ;
```

Get the first tuple in response to a query. The response must include the following (at least) :

- NULL or FOUND to indicate either.
- relation_id
- tuple_id
- attribute-value-pairs
 - attribute name
 - type
 - number of characters
 - value

Note that it is not necessary to send all the attributes at once. A minimal set consists of at least the following <TBD>

Final Scientific Report: RaPIER Project (Contract No. N00014-85-C-0666)

```
procedure SB_process(in SB_query : SB_QUERY_TYPE;
                    out SB_response : SB_PARSED_QUERY_TYPE;
                    )
    returns (SUCCESS | FAILURE);
```

Scan, parse, and execute an SB query. For the response, see procedures SB_parse and SB_execute.

```
procedure SB_begin_accept_module(in module_name : STRING;
                                out tup_id : TUPLE_IDS
                                )
    return (SUCCESS | FAILURE);
```

Create a new module by first creating a tuple for it; return the tuple id of the new tuple.

```
procedure SB_get_attr(in tup_id : TUPLE_IDS;
                     in att_name : STRING;
                     out SB_response : SB_RESPONSE_TYPE
                     )
    returns (NULL | FOUND) ;
```

Get the named attribute from the specified tuple.

```
procedure SB_get_code(in procedurename : STRING;
                     out SB_response : SB_RESPONSE_TYPE
                     )
    returns (SUCCESS | FAILURE);
```

Transmit the source code of the named module onto RaPIER.

```
procedure SB_get_next_attribute(in tup_id : TUPLE_IDS;
                               out SB_response : STRING
                               )
    returns (NULL | FOUND | END_OF_TUPLE);
```

Get the next attribute of a tuple.

```
procedure SB_get_next_tuple(in relation_id : RELATION_IDS;
                            in number : INTEGER;
                            in direction : BOOLEAN;
                            out SB_response : TUPLE_ID
                            )
    returns (NULL | FOUND) ;
```

Software Base - Classification Scheme

Get the next tuple of this relation. Note that because queries may be nested, it is necessary to keep track of relation and tuple ids. The variable "number" indicates the number of tuples to returned and while "direction" indicates which direction to go relative to the position of the cursor on the database buffer. (For example "1" indicates that the tuples above the cursor are to be retrieved while "0" indicates that the tuples below the cursor are to be retrieved. It is up to the RAPIER system to maintain its own currencies.

```
procedure SB_put_attribute(in tup_id : TUPLE_IDS;
                           in att_name : STRING;
                           in att_value : ATT_TYPE
                           )
```

Put the attribute value of the named attribute into a tuple.

```
procedure SB_get_tuple(in relation_id : RELATION_IDS;
                       in tuple_name : TUPLE_ID;
                       out SB_response : TUPLE_ID
                       )
  returns (SUCCESS | FAILURE);
```

Get a the named tuple from the specified relation. This is almost a direct access.

```
procedure SB_open_relation(in relation_id : RELATION_IDS;
                           out SB_response : SB_RESPONSE_TYPE
                           )
  returns (NULL | FOUND) ;
```

Get ready the results of a previous retrieval for transmission to RAPIER.

```
procedure SB_parse(in SB_query : SB_QUERY_TYPE;
                   out SB_response : SB_PARSED_QUERY_TYPE
                   ) returns (SUCCESS | FAILURE);
```

Scan and parse an SB query. Return a success or failure code to indicate either; this code is followed by either a string of error messages or a string of tokens which constitute an internal form of an SB query in pre- or postfix.

```
procedure SB_pattern_match(in pattern : STRING;
                           in tup_id : TUPLE_IDS;
                           in att_name : STRING;
                           )
  returns (MATCH | NO_MATCH) ;
```

Match a given string against the value of an attribute.

```
procedure SB_store_code(in module_name : STRING;
                        in tup_id : TUPLE_IDS;
                        in code : ADA_CODE;
                        out SB_response : SB_RESPONSE_TYPE
                        )
    returns (SUCCESS | FAILURE);
```

Store a piece of Ada code and return a tuple id of the tuple describing the code.

```
procedure SB_end_accept_module(in modulename : STRING;
                              out SB_response :SB_RESPONSE_TYPE
                              RETURNS (SUCCESS | FAILURE);
```

End this transaction and inform RaPIER if the module was entered.

```
procedure SB_enter_specification(in specs: SB_SPECIFICATION_TYPE;
                                in modulename : STRING;
                                out response : SB_RESPONSE_TYPE;
                                returns (SUCCESS | FAILURE);
```

Enter the specification for a given module.

SECTION 6

REUSABILITY GUIDELINES FOR ADA

This section presents metacharacteristics, characteristics, and guidelines for writing reusable Ada source code. This material is the result of work on Task H1.3 and Honeywell-funded work. We present the material as a self-contained guidebook, with its own table of contents and bibliography.

6.1 PROBLEM STATEMENT

Both software production costs and the amount of new software produced annually are skyrocketing. In 1980, the U.S. Department of Defense (DoD) spent over \$3 billion on software. By 1990, their expenses are expected to grow to \$30 billion/year [HOROWITZ84]. If current development trends continue, future costs will be increased even more by unreliable software, software delivered late, and continuing maintenance problems.

Today's software needs outpace our ability to produce it, as shown by the backlogs in MIS departments nationwide, and needs are growing each year [STARS83]. There is and will continue to be a serious shortage of qualified programmers to meet these needs. One might expect productivity increases for programmers to make up for at least a part of this shortage. However, software development has seen relatively small year-to-year productivity increases as contrasted with dramatic increases in hardware fabrication [HOROWITZ84]. We feel that a key to significant gains in programmer productivity lies in the area of software reuse. Reuse makes particularly good sense since the cost of software is an exponential function of its size. Halving the amount of new software built will more than halve the cost of building the software that we need [JONES84].

Software reuse is an important part of the RaPIER project for many of the same reasons it is important to software productivity increases in general. Remember, one of RaPIER's main goals is "...to develop a prototype engineering environment [that will] provide tools and techniques for developing modifiable prototypes quickly and inexpensively...". The approach to achieving this goal is to build prototypes from reusable software parts. It is the characteristics of these reusable software parts that will provide the modifiability, and the rapid and inexpensive development of prototypes that RaPIER requires.

To date, no adequate characterization of what makes software reusable exists. It is quite common to read unmeasurable, qualitative admonitions as to what makes software reusable and/or specific examples of software that is claimed

to be reusable. However, these admonitions (or "metacharacteristics") and software examples are not enough. Measurable characteristics of reusable software are needed as well as specific guidelines to implement them in source code. Only through use of these characteristics and guidelines can the full potential of reusability be achieved.

6.2 OUTCOME

The RaPIER project has developed Version 1.0 of "A Guidebook For Writing Reusable Source Code in Ada (R)." This guidebook contains three reusability metacharacteristics, fifteen measurable characteristics that realize the metacharacteristics, and 63 guidelines for implementing these characteristics in Ada source code. Guidebook chapters are organized to follow the Ada Language Reference Manual [DOD83]. Version 1.0 of the guidebook contains selected chapters covering all major Ada program units, program structure, compilation issues, and visibility rules. Example Ada modules that were written following the guidelines are also provided. This guidebook provides the RaPIER project with a basis to begin writing reusable Ada software parts to be used in its prototyping system.

6.3 REUSABILITY GUIDEBOOK

The following pages contain version 1.0 of "A Guidebook for Writing Reusable Source Code in Ada (R)." This guidebook will also appear separately as a technical report.

A GUIDEBOOK FOR WRITING REUSABLE SOURCE CODE IN ADA (R)

Version 1.0

R. St. Dennis

Honeywell Inc.
Computer Sciences Center
1000 Boone Avenue North
Golden Valley, Minnesota 55427

March 1986

(1) Ada is a registered trademark of the U.S. Government (AJPO)

This work was supported in part by the Office of Naval Research under contract number N00014-85-C-0666.

blank back page

CONTENTS

	Page
Preface	81
I: REUSABILITY, CHARACTERISTICS OF REUSABLE SOFTWARE, AND ADA	83
I-1 Introduction	83
I-1.1 Reusability -- A Definition	84
I-1.2 Our Approach to Achieving Reusability	86
I-1.3 Other Approaches to Reusability	87
I-1.4 Organization of this Guidebook	88
I-2 Characteristics of Reusable Software	89
I-2.1 Criteria for Reusability Characteristics	89
I-2.2 List of Characteristics	89
I-3 The Ada Programming Language and Reusability	98
I-3.1 Ada Design Goals	98
I-3.2 Ada and Reusability	98
II: GUIDELINES FOR WRITING REUSABLE ADA SOFTWARE	101
II-1 Introduction	101
II-2 Lexical Elements	102
II-2.1 Ada Summary	102
II-2.2 Guidelines	102
II-2.2.1 Lexical Elements in General	102
II-2.2.2 Pragmas	102
II-2.3 Guideline/Characteristic Cross Reference	102
II-3 Declarations and Types	103
II-3.1 Ada Summary	103
II-3.2 Guidelines	103
II-3.2.1 Declarations and Types in General	103
II-3.2.2 Object and Named Number Declarations	103
II-3.2.3 Types and Subtypes	103
II-3.2.3.1 Scalar Types	103
II-3.2.3.2 Composite Types	103
II-3.2.3.3 Access Types	103
II-3.2.4 Derived Types	103
II-3.2.5 Declarative Parts	103
II-3.3 Guideline/Characteristic Cross Reference	103
II-4 Names and Expressions	104

CONTENTS (cont)

	Page
II-4.1 Ada Summary	104
II-4.2 Guidelines	104
II-4.2.1 Names	104
II-4.2.2 Expressions	104
II-4.2.2.1 Operators	104
II-4.2.2.2 Type Conversions	104
II-4.2.2.3 Qualified Expressions	104
II-4.2.2.4 Allocators	104
II-4.3 Guideline/Characteristic Cross Reference	104
II-5 Statements	105
II-5.1 Ada Summary	105
II-5.2 Guidelines	105
II-5.2.1 Simple Statements	105
II-5.2.2 Compound Statements	105
II-5.3 Guideline/Characteristic Cross Reference	105
II-6 Subprograms	106
II-6.1 Ada Summary	106
II-6.2 Guidelines	106
II-6.2.1 Subprograms in General	106
II-6.2.2 Subprogram Declarations	108
II-6.2.3 Subprogram Bodies	109
II-6.2.4 Formal Parameter Modes	115
II-6.2.5 Subprogram Calls, Default Parameters and Parameter Associ- ations	116
II-6.2.6 Overloading of Subprograms	117
II-6.3 Guideline/Characteristic Cross Reference	119
II-7 Packages	121
II-7.1 Ada Summary	121
II-7.2 Guidelines	121
II-7.2.1 Package Structure	122
II-7.2.1.1 Package Specifications and Declarations	125
II-7.2.1.2 Package Bodies	127
II-7.2.2 Private Type and Deferred Constant Declarations	129
II-7.2.3 Additional Considerations	131
II-7.3 Guideline/Characteristic Cross Reference	131
II-8 Visibility Rules	132
II-8.1 Ada Summary	132
II-8.2 Guidelines	132
II-8.2.1 Use Clauses	132
II-8.2.2 Renaming Declarations	134
II-8.2.3 Package Standard	136
II-8.3 Guideline/Characteristic Cross Reference	137

CONTENTS (cont)

	Page
II-9 Tasks	139
II-9.1 Ada Summary	139
II-9.2 Guidelines	139
II-9.2.1 Tasks in General	139
II-9.2.2 Task Declarations	140
II-9.2.3 Task Bodies	141
II-9.2.4 Task Types and Task Objects	146
II-9.2.5 Entries, Entry Calls, and Accept Statements	147
II-9.2.6 Delay Statements, Duration, & Time	148
II-9.2.7 Select Statements	148
II-9.2.8 Priorities	149
II-9.2.9 Abort Statements	149
II-9.3 Guideline/Characteristic Cross Reference	150
II-10 Program Structure and Compilation Issues	151
II-10.1 Ada Summary	151
II-10.2 Guidelines	151
II-10.2.1 Compilation Units - Library Units	151
II-10.2.1.1 Directly Reusable Parts	152
II-10.2.1.2 Indirectly Reusable Parts	157
II-10.2.2 Context Clauses	157
II-10.2.3 Subunits of Compilation Units	158
II-10.2.4 Order of Compilation	159
II-10.2.5 The Program Library	160
II-10.2.6 Elaboration of Library Units	160
II-10.3 Guideline/Characteristic Cross Reference	160
II-11 Exceptions	162
II-11.1 Ada Summary	162
II-11.2 Guidelines	162
II-11.2.1 Exception Handlers	162
II-11.2.2 Raise Statements	162
II-11.2.3 Exception Handling	162
II-11.2.4 Suppressing Runtime Checks	162
II-11.3 Guideline/Characteristic Cross Reference	162
II-12 Generic Units	163
II-12.1 Ada Summary	163
II-12.2 Guidelines	163
II-12.2.1 Generic Declarations	163
II-12.2.1.1 Generic Formal Objects	172
II-12.2.1.2 Generic Formal Types	172
II-12.2.1.3 Generic Formal Subprograms	173
II-12.2.2 Generic Bodies	175
II-12.2.3 Generic Instantiations	176
II-12.3 Guideline/Characteristic Cross Reference	177

A Guidebook for Writing Reusable Source Code in Ada

CONTENTS (cont)

	Page
I-13 Representation Clauses and Implementation Dependent Features	178
I-13.1 Ada Summary	178
I-13.2 Guidelines	178
I-13.2.1 Representation Clauses	178
I-13.2.1.1 Length, Enumeration, Record, and Address Clauses	178
I-13.2.1.2 Change of Representation	178
I-13.2.2 Implementation Dependent Features	178
I-13.2.2.1 The Package System	178
I-13.2.2.2 Machine Code Insertions	178
I-13.2.2.3 Interface to Other Languages	178
I-13.2.2.4 Unchecked Programming	178
I-13.3 Guideline/Characteristic Cross Reference	178
II-14 Input-Output	179
II-14.1 Ada Summary	179
II-14.2 Guidelines	179
II-14.2.1 External Files and File Objects	179
II-14.2.2 Sequential and Direct Files	179
II-14.2.3 Text Input-Output	179
II-14.2.4 Exceptions in Input-Output	179
II-14.2.5 Low-Level Input-Output	179
II-14.3 Guideline/Characteristic Cross Reference	179
A Appendix: List of Guidelines	180
B Appendix: Guidebook-Wide Characteristic/Guideline Cross Reference	185
C Appendix: Example Ada Modules	187
C.1 Description of Example Modules	187
C.2 Guidelines Illustrated	189
C.3 Ada Modules	190
D Appendix: Glossary	200
Bibliography	201

PREFACE

This guidebook was produced as part of the RaPIER (Rapid Prototyping to Investigate End-user Requirements) project at the Honeywell Computer Sciences Center. It is version 1.0 of a guidebook scheduled for completion in the third quarter of 1986 and contains selected chapters from the final guidebook. The purpose of the RaPIER project is to develop a methodology and automated support (the RaPIER system) for prototyping embedded computer systems and software. The project's co-equal goals are: (1) to develop technology for building and using prototyping in system and software development, and (2) to insure smooth transfer of that technology to Honeywell operating divisions and the U. S. government. The RaPIER system consists of a repository of reusable code and a front-end workstation for constructing and executing prototypes.

I wish to thank a number of people who helped me during the writing of this guidebook, most notably Elaine Frankowski, Emmanuel Onuegbu, and Paul Stachour. These people were my coauthors on a paper entitled, "Measurable Characteristics of Reusable Ada(R) Software," which will appear in an upcoming issue of Ada Letters. Specifically, Elaine provided excellent technical insight into the possibilities of and obstacles to reusability of source code, organization to my sometimes wandering ideas, encouragement, advice, moral support; in short, everything I needed to write a guidebook of this size. Emmanuel provided consultation support in the area of software classification and software bases. This gave me necessary insight into systems to store and examine reusable source code and the requirements these systems may place on the code itself. Paul, in his role as an "Ada Language colleague," provided me with ideas for guidelines and good advice.

I would also like to thank other members of the Honeywell RaPIER Project and Software Development Technology Department: Curt Abraham, Lai King Mau, and Mark Spinrad for their reviews and general support, and Dede Schmidt for her excellent clerical support.

blank back page

SECTION I:

REUSABILITY, CHARACTERISTICS OF REUSABLE SOFTWARE, AND ADA

I-1 INTRODUCTION

Both software production costs and the amount of new software produced annually are skyrocketing. In 1980, the U.S. Department of Defense (DoD) spent over \$3 billion on software. By 1990, their expenses are expected to grow to \$30 billion/year [HOROWITZ84]. If current development trends continue, future costs will be increased even more by unreliable software, software delivered late, and continuing maintenance problems.

Today's software needs outpace our ability to produce it, as shown by the backlogs in MIS departments nationwide, and needs are growing each year [STARS83]. There is and will continue to be a serious shortage of qualified programmers to meet these needs. One might expect productivity increases for programmers to make up for at least a part of this shortage. However, software development has seen relatively small year-to-year productivity increases as contrasted with dramatic increases in hardware fabrication [HOROWITZ84]. We feel that a key to significant gains in programmer productivity lies in the area of software reuse. Reuse makes particularly good sense since the cost of software is an exponential function of its size. Halving the amount of new software built will more than halve the cost of building the software that we need [JONES84].

[JONES84] reports that the required functionality of a large amount of all software produced is common or redundant. This would lead us to believe that software reuse is a very common practice today. However, while software is being reused, this is only true in limited application areas and cases [STANDISH84], [HOROWITZ84]. Why isn't reusability more prevalent?

In order to achieve reusability, a number of conditions must be met. Most importantly, an adequate characterization of what makes software reusable must exist. To date it is quite common to read unmeasurable, qualitative admonitions as to what makes software reusable and/or specific examples of software that is claimed to be reusable. However, these admonitions (or "metacharacteristics") and software examples are not enough. Measurable characteristics of reusable software are needed. These characteristics can form the basis for guidelines that programmers can follow to write reusable software and to determine whether software written by others is reusable.

A Guidebook for Writing Reusable Source Code in Ada

This guidebook is the result of an ongoing research effort at the Honeywell Computer Sciences Center (CSC) to define measurable characteristics of reusable software as well as guidelines for implementing them in Ada.

I-1.1 REUSABILITY -- A DEFINITION

[WEBSTER77] defines "reusable" as "capable of being used again or repeatedly;" "reuse" as "to use again esp. after reclaiming or reprocessing..."; and "further or repeated use." [KERNIGHAN84] defines reusability as "...any way in which previously written software can be used for a new purpose or to avoid writing more software." In this guidebook, we address reusability of source code which may involve some modification of the code.

We propose these metacharacteristics of reusable software: (1) Candidate software for reuse must be able to be found; (2) Once found, software must be understood enough to be reused; (3) Once found and understood, software must be feasible to reuse. Software which is feasible to reuse (a) is built for reuse, (b) is fit for reuse (i.e., a "plug-compatible" part), and (c) displays conceptual clarity or appropriateness. Findable, understandable, reuse-feasible software is more economical to reuse than to recreate. Chapter I-2 posits 15 software characteristics that realize these metacharacteristics.

These are the assumptions that underlie our discussion of the metacharacteristics and characteristics:

1. Reusable software is well-engineered software, designed and coded according to the best software engineering practices known today. Badly designed algorithms and implementations should not be used even once, a fortiori they should not be perpetuated through reuse. In addition, badly engineered software will not meet the reusability characteristics and guidelines proposed in this and the following chapter. Therefore all the accepted software engineering guidelines for "good" design and code also apply to reusable code.
2. Reusable software is stored on some computer under the control of a database (or software-base) management system. If the software resides in a simple file system, or is kept only on paper, some of the following discussion does not apply. However, parts of it are valid no matter how candidate software is stored and managed.
3. Software can be reused with or without modification. It is preferable to reuse a software part as a "black-box," tailoring it only by external data: tables, parameters, and so forth. However, until we become experienced at designing reusable software, reuse may involve some code modification. Therefore all the accepted software engineering guidelines for modifiable code also apply to reusable code.

Let us examine the metacharacteristics in some detail.

1. Candidate software for reuse must be able to be found.

Findable software must comprise both code and specification. At a minimum, the specification tells users what a software part does, thus allowing them to decide whether it meets their functional needs. This information may be presented formally or informally. A specification may describe attributes of the software part such as author, hardware dependencies, execution time on a particular configuration, and so forth, which further assist users in deciding what software is appropriate. If the specification is formal it is machine manipulable and thus a software base management system may use it in satisfying queries and "browses." See Section 5 of [RAPIER86]. Even if the formal specification is only a compilable procedure header with an evocative name and descriptive parameter names and types, it serves human communication needs as well as being executable.

The apparatus for storing and managing software contributes greatly to its findability. That apparatus includes a software base management system and intelligent schemes for classifying software so that searches into the software base are successful without being frustratingly long. The need to classify software to facilitate browsing in an unfamiliar repository puts demands on the model under which the software is developed (e.g., functional, object-oriented), and the nature of the software part itself (e.g., highly cohesive [BERGLAND81]).

It must be significantly less costly to find software and reuse it than to recreate it. Both the specifications and the apparatus for managing the reusable software must support relatively low (human and machine) overhead for storing software and searching for it.

2. Once found, software must be understood enough to be reused.

This requirement involves both the software part's specification and, if its code is to be modified, the way in which it is coded. All the problems with formal or informal specifications [FREEMAN83] in general apply to specifications of reusable software also: resistance to using formal specifications, inability to write formal or informal specifications that accurately say what a piece of software does, the difficulties in specifying performance, reliability, and so forth. As these problems are solved, the solutions must be adopted for specifying reusable software. In addition, there are judgments to be made about what attributes of a software part re-users need to know in order to decide whether the software meets their needs.

If the software is to be modified, it must be engineered so that re-users can examine the code and make changes that do not introduce errors or unwanted side effects, and that do make the desired alterations.

3. Once found and understood, it must be feasible to reuse the software.

Software that can be reused

A Guidebook for Writing Reusable Source Code in Ada

- o is built for reuse - constructed under the constraint that it will be reused. This constraint will cause builders to eschew hidden side effects, assure that the component does not reference global data structures or hardware devices that may or may not be present when it is reused, provide as good a specification as they can, and so forth.
- o is fit for reuse (i.e., is a "plug-compatible" part) - composable with other code in such a way that it neither interferes with that other code nor allows itself to be interfered with. This constraint will lead to software that, for example, includes scaffolding that can be used during a "build" to test the software part in combination with other parts, that handles errors in a standard way, or that makes extensive use of parameters or other language constructs for modifiability.
- o displays conceptual clarity or appropriateness - presents a useful abstraction (such as a table, a database, a sensor or a stack) at an "appropriate" level. This constraint should lead to software parts built under some model such as the Smalltalk [GOLDBERG83] object model, and to software reuse under a system model such as "software as simulation" [MacLENNAN85] or the Lisp flavor model [SYMBOLICS84].

Each of the software characteristics proposed in Chapter I-2 is a means of achieving one (or some) of these metacharacteristics. Figure I2-1: Reusability Characteristics Realize the Metacharacteristics, provided in Chapter I-2, relates each of the proposed characteristics to the metacharacteristics it promotes.

I-1.2 OUR APPROACH TO ACHIEVING REUSABILITY

Our approach to reusing source code centers around reusable components, written as Ada packages, classified for both browsing and retrieval, and residing in a library or software base. See Section 5 of [RAPIER86]. We believe that the features of the Ada language combined with a set of software design and coding guidelines supporting measurable characteristics of reusability that achieve the metacharacteristics defined above will enable creation and reuse of software in a manner not possible with most other languages and systems. These guidelines will constrain how Ada software is written for the sake of reusability. High-level features of the Ada language that support reusability are discussed in Chapter I-3.

Companion work at Honeywell's Computer Sciences Center is also addressing the organization and composition principles that will provide a framework for reuse of components. A classification of components according to behavior has been proposed in Section 5 of [RAPIER86]. Program composition using an adaptation of the operational paradigm for program design has also been proposed in Section 3 of [RAPIER86]. A high-level language for composing programs of components drawn from a the software base using a Prototype System

Description Language (PSDL) is being designed by International Software Systems, Inc. (ISSI) [ISSI86]. So the characteristics and guidelines in this guidebook fit into an overall approach to reusability.

There are technical issues to be resolved before this approach can be applied. Technology must be developed in the areas of:

1. the reusable code itself,
2. information about the code, such as its documentation, specification, classification, the recording of design information, and its attributes.
3. the apparatus for archiving and managing reusable code, and
4. defining a development life cycle that permits and promotes code reuse.

This guidebook attempts to deal with the first two issues, assuming an archiving apparatus. The life cycle issues are left to others.

One can design and write code and simply claim it is reusable, but if it does not satisfy the metacharacteristics mentioned above and measurable characteristics defined in Chapter I-2 it will not be reused. As we stated earlier, measurable characteristics form the underlying basis for guidelines for writing reusable software. These guidelines are essential for two reasons. First, they define reusability in terms that can be understood and followed by software developers. Second, these guidelines provide a means to control development of reusable software (i.e., promote a standard reusable software product that is amenable to automatic retrieval and systematic modification/use and is understandable as well). At this point, we suspect that standards for developing reusable software are more important than individual programming styles.

I-1.3 OTHER APPROACHES TO REUSABILITY

[BIGGERSTAFF84] groups the various approaches to reusability into two broad categories. The first group consists of those approaches that emphasize the accumulation, organization, and composition of components, while the second group consists of approaches that emphasize the generation of the target program. For the second group, the authors noted that it is the patterns that generate or transform programs that get reused; pre-existing components do not necessarily reappear intact (if at all) in the target programs. Very High-Level Languages (VHLLs), application generators, problem-oriented languages (POLs), and program transformation systems belong to this second group.

Our approach to reusability falls squarely into the first group: we will reuse code. Other approaches in this group emphasize the reuse of design or specifications. In order to accumulate code, we must first design and build a repository or software base. The characteristics of the software to be stored in such a repository must make it attractive to users. This is the motivation for suggesting guidelines for writing reusable code.

A Guidebook for Writing Reusable Source Code in Ada

I-1.4 ORGANIZATION OF THIS GUIDEBOOK

This guidebook is organized into two sections. Section I introduces the reader to the concepts and state-of-the-art in software reusability, presents our approach to reusability, and surveys other approaches. Our approach is described in terms of characteristics of reusable software and how the Ada programming language in a high-level way can be used to implement these characteristics. Section II presents specific guidelines on how the above-mentioned characteristics of reusable software identified can be realized in Ada code. It is meant to be a reference manual. To this end, it is organized in chapters that follow the Ada Language Reference Manual [DOD83]. Appendix A provides a list of all guidelines appearing in Section II. Appendix B provides a cross reference between the reusability characteristics we posit in Section I and these guidelines. Appendix C contains example Ada modules written following our reusability guidelines. Appendix D contains a glossary of terms.

AD-A166 353

JOINT PROGRAM ON RAPID PROTOTYPING RAPIER (RAPID
PROTOTYPING TO INVESTIGA. (U) HONEYWELL INC GOLDEN
VALLEY MN COMPUTER SCIENCES CENTER 28 MAR 85

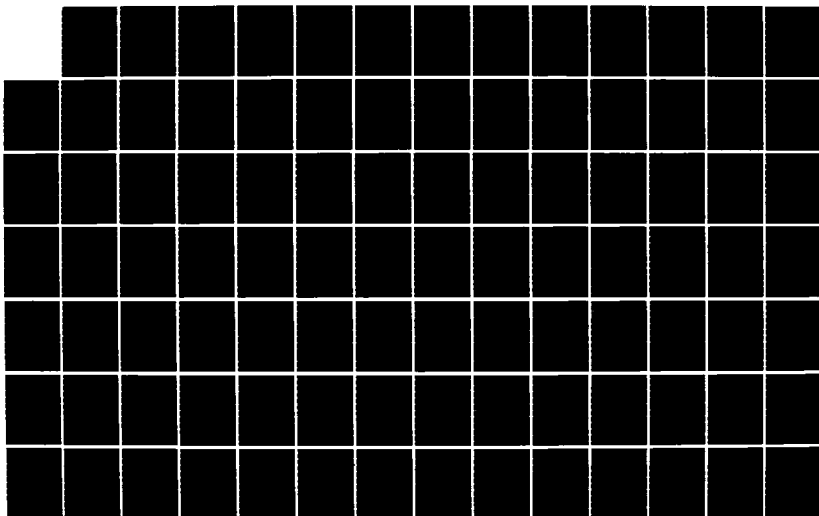
2/4

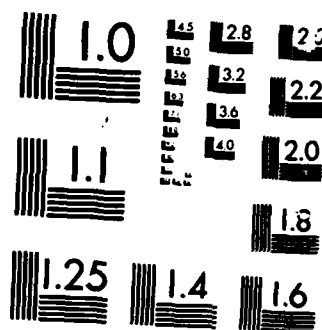
UNCLASSIFIED

N00014-85-C-0666

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART

I-2 CHARACTERISTICS OF REUSABLE SOFTWARE

I-2.1 CRITERIA FOR REUSABILITY CHARACTERISTICS

The reusability metacharacteristics (see Chapter I-1) are qualitative "good practice" admonitions. In general, they are not measurable or achievable without further guidance. In contrast, the characteristics are measurable or judgable qualities that software should possess in order to meet the metacharacteristics. We have proposed characteristics that are statically measurable or judgable today or will be measurable/judgable once we have more experience with reusable software. For example, today we can measure if software is free from hidden side effects. However, we cannot judge whether software has the right balance between generality and specificity. Only when software has been reused for some time, we will be able to judge this quality.

The characteristics listed below are also reuse-specific; using them will produce software that is designed and coded a priori for reuse. The "good" software practices mentioned in Chapter I-1 will contribute to reuse but will not specifically make software reusable.

This guidebook only briefly discusses an important aspect of reusability: domain or application specificity. We expect that application specificity will be a major factor in enabling software reuse [FRANKOWSKI85b]. We further expect that some applications permit more reuse than others; an extreme example is that more business data processing software can be reused [LANERGAN84] than can embedded software for avionics systems. However, just as all software intended for reuse must be built using good software engineering practices (see Chapter I-1), it must be built using application neutral basic reusability guidelines in addition to application specific guidelines. The characteristics listed below are those underlying guidelines for reusability across application areas.

I-2.2 LIST OF CHARACTERISTICS

Characteristics of reusable software include:

1. Interface is both syntactically and semantically clear [STANDISH84].

In order to reuse a piece of software correctly, its interface must be understood. If one considers an interface as a socket to which client programs can connect, then each facet of the socket must be clearly defined. The syntax must not only be correct, but should also convey some meaning. Mnemonic names should be used in order to quickly convey the functionality of an interface. For example, an interface called "draw_circle" is certainly more descriptive than one called "d_c;" a parameter named "radius" is also more descriptive than one named "rd." Also, parameters should be typed, checked (preferably at compile time),

and constrained to satisfy the domain of the problem to be solved. For example, if an object can reasonably be assumed to take on values of 0 and 1, it should be of this type rather than an integer.

A combination of language features and documentation constructs should be used to convey the semantics of the interface to the user. Each parameter should be defined with an appropriate mode (e.g., in, out, etc.) and should have a brief explanation of its function (e.g. "plant_location is the name of the city where the plant is located"). The user should be informed of all the assumptions that a component makes in order to execute correctly. For example, a binary search function assumes that the list it searches is ordered; an explicit statement of that assumption enables the user to reuse such a function only in the appropriate contexts. In this way, the interface definitions and their documentation constitute natural language functional specifications (though degenerate) that aid users to judge the reusability of a component.

The more visible the description of a software module's interface, the easier the module will be to reuse. All the interfaces exported and imported by a module should be clearly described in the documentation for that module.

2. Interface is written at appropriate (abstract) level.

Reusable code should be written so that its interface is at a level appropriate for its function. For example, a routine that handles tables of type `table_type` should be passed an object of type `table_type`, not subobjects corresponding to the table entries. Clarity is sacrificed if the interface is not at the appropriate level. As another example, consider a function that makes decisions based on the information contained in a project database. Since a corporation president's view of the project is at a higher level of abstraction than that of a middle-line manager, having a different interface for each user is more appropriate than having one interface through which all the details concerning the project must pass.

Another aspect of level involves nesting of language constructs. It is easier to reuse a component whose interface is no deeper than one nested level within the declarative part of a compilation unit. Single-level nesting can make classification of reusable software simpler and makes the software easier to understand since its "environment" or context is not over-complicated by language visibility rules and hiding.

Whenever different users need to reuse the same function at different levels, it is more reasonable to have packages, each dedicated to one level of abstraction, than to have one package in which all levels of abstraction are muddled together.

3. Component does not interfere with its environment.

Characteristics of Reusable Software

Interference is unexpected behavior or behavior that cannot be "rolled-back." Any well-defined software should not interfere with its environment: it should generally not change global data, be free from hidden side-effects, have a clear and predictable effect on its environment, and should minimize assumptions it makes about its environment. Non-reusable software, with its one-time environment, can violate this guideline and still be effective (though maintenance, a form of reuse, may be adversely affected).

What is the environment for a reusable part? It may be clear what its environment was as part of the system for which it was originally developed. However, in general, its environment is the sum total of all possible "situations" in which it may be used now and in the future. Because this set of situations may not be totally specifiable, it is essential for software that is to be reusable to meet this guideline.

To be specific, reusable code should be written using language features that make clear exactly what the code expects from the environment, what effects it will have on its environment, and how these effects will be realized. These expectations and effects should always be documented by comments in the code. Communication of code effects should be done using parameters or function values. Language features that restrict data to its intended use should be utilized (e.g., mode of parameters in subprograms, etc.). Use of visibility rule/naming constructs that are convenient in isolation but can cause ambiguities when used together with other code (i.e., in a different environment) should be minimized.

4. Component is designed as object-oriented; that is, packaged as typed data with procedures and functions which act on that data.

An object orientation to code involves mapping of "solutions" to our human view of the "problems" the software is trying to solve [BOOCH83]. Our human view involves objects, attributes of these objects, and operations on objects expressed in a noun/verb sense in English. An object orientation to software aids understandability since solutions to problems are expressed in our "human terms."

Reusable software should act on objects explicitly. What we are advocating here is a clear definition and method of "acting" on objects. All actions or operations on objects should be defined as subprograms (or their equivalent) with the objects as parameters. Furthermore, the objects or at least their types should be "packaged" as close to the definition of the operations on them as possible. Ideally, they should be packaged together to ease location, reference, and use. To promote reusability it is better not to use global data that is changed implicitly by routines to which it is visible but to pass the data to routines as parameters making it explicit that (1) these routines are actors/operators on the data and (2) this is just how this data will be treated (e.g., as input only, as a constant, and so forth).

Based on Section 5 of [RAPIER86], we will define operations on data in context as implementations of behaviors that characterize objects, the objects being defined by the set of all behaviors associated with them.

5. Actions based on function results are made at the next level up.

A software module produces some result or effect. The module must allow its client to initiate all further actions, calculations, and decisions based on the result or effect. Both producing and using an effect or result makes for highly-coupled software that can be misunderstood or produce mysterious effects.

Reusable software needs to have a definite predictable function based on a well-defined set of inputs/environmental parameters. Code written to always pass results up to its caller is predictable. For example, if a routine needs to use a tape drive to perform its function and the drive is not available, it should report back this condition, not try to handle it by printing a "tape drive not available" message. Code that uses this routine should expect to handle this condition and may treat it as an error or simply ignore it. Thus, this guideline promotes both understandability and fitness for reuse (i.e., composability or plug compatibility).

6. Component incorporates scaffolding for use during "building phase."

[BENTLEY85] defines software scaffolding as "...temporary programs and data that give programmers access to system components." This scaffolding is analogous to scaffolding around a building under construction which gives workers access to the building they would not normally have. Another analogy is built-in test equipment in hardware. Scaffolding in software development isn't usually delivered to the customer but can be of immense help during debugging, testing, and integration. Scaffolding produces software that is built for reuse.

For the sake of reusability, scaffolding can provide valuable help in integrating a piece of software with others and/or using it for the "first few times." Scaffolding may include print routines to show program behavior, special error-handling routines, routines to check and view interfaces, parameterization to minimize the amount of modification required for use, and so forth.

7. Separate the information needed to use software, its specification, from the details of its implementation, its body.

The specification of a software component is what needs to be visible to the user. The details of the implementation are not directly necessary; such details may turn out to be superfluous, confusing, and may be used incorrectly. Implementation details should therefore be separated from a component's specification and hidden from the user. A typical example is the separation of the body of an Ada package from its specification; users do not need to know about the body of the package in order to use it. There may be cases where implementation information is needed; for

example, a database system might inform the user about how to choose access strategies in order to speed up retrieval. Such information can still be incorporated into a specification. One should therefore use language features to separate specification of function from its implementation as well as to limit access to representation details of data used in communicating with the software.

8. Component exhibits high cohesion/low coupling [BERGLAND81].

Functionally cohesive software modules lend themselves to understandability and thus to reusability. High coupling to other modules limits a component's use in isolation since the modules to which it is coupled effectively form an environment which is necessary for its compilation and execution. Reuse of the component in another environment may be desired but not possible because of its high connection with its original environment.

One effective means of achieving high cohesion is by using an object-oriented program design methodology. The object oriented paradigm provides actors, each with its own set of applicable operations. An object oriented design strategy makes it natural to detect low cohesion; for example, a screen manager that performs namespace management functions is obviously not functionally cohesive. Also, the higher the cohesion of a module, the easier it will be to classify.

9. Component and interface are written to be readable by persons other than the author.

The need for interface readability stems from the fact that a component's interface is part of its documentation, and that documentation is used by people. Code readability is related to the need for modifying code. Though modifying a module by changing its code is less desirable than modifying it with external data, code modification will be a fact of life until the software development community becomes experienced in writing software that can be reused with only external modifications.

10. Component is written with the right balance between generality and specificity [MATSUMOTO84].

Components that are too specific do not lend themselves to reuse by a large user community. A sort package that works on character strings as well as numbers can be reused by more users than a package that only sorts integers. There is a need, however, to maintain a balance between generality and specificity because there is often some price to pay for generality. One cost of over-generality is performance, since extra processing is required in order to customize the general module to a specific application. Another cost is the slower learning curve: the more general the component, the more knowledge that is required to use it.

One effective way to achieve generality is through parameterization. We therefore encourage the use of language features that allow increased parameterization [COGUEN84]. A module with parameters for all possible

design choices is both general and tailorable to many situations in an application area. It can probably be reused without modifying any code. A highly parameterized module with default values for most of its parameters is particularly convenient to reuse, since reusers need to consider only differences between the default situation and theirs and give values to only a few parameters, rather than define the complete situation by providing values for every parameter.

11. Component is accompanied by sufficient documentation to make it findable.

An environment which encourages software reuse is usually equipped with (at least) a means of cataloging reusable components and a means of interrogating the catalogue. Modern database management systems, relational databases for instance, are suitable for storing and cataloguing components since they provide ad-hoc query facilities for retrieving components based on the values of some keys (information) associated with the components. Some of the pertinent information that should be stored along with a component [YEH84, AMANO84] include purpose, revision history, resources required (e.g. other modules), languages, operating system, runtime utilities, I/O devices, automatic interrupts affecting module execution, memory requirements [MATSUMOTO84], and performance characteristics. Writers of the reusable component should therefore fill in as much information as a software base needs to successfully include the component in its catalogue. They should also include enough textual documentation in the code itself to enable those who look at the code to understand or locate the portions of interest.

12. Component can be used without change or with only minor modifications.

Ideally, code can be reused with no changes required. This implies that it can be understood and provides the required functionality, interface, and possibly performance characteristics. Developers of reusable software parts should use language features that support parameterization in the code. (See characteristic 10 above). If some changes will inevitably be required in code in order for it to be reused, developers should isolate and identify those things that are expected to need changing.

Quite commonly, code is reused by grouping it together with other software. Developers should use language features that facilitate this grouping. For example, developers should exploit language features to reference previously-compiled code instead of having to combine code with a text editor and recompile it before reuse.

13. Insulate a component from host/target dependencies and assumptions about its environment; Isolate a component from format and content of information passed through it which it does not use.

A reusable component should not depend on an implicit environment. Contact between the component and the world outside should occur only through explicit parameters and explicitly invoked subprograms. This guideline rules out "bit-flicking" or machine-dependent optimization and encourages the use of higher-level languages for coding.

All the assumptions made by the component must be explicitly stated as well. For example, a type conversion program may assume that whenever it has to convert an instance of type A into a new instance of type B, there does not have to be any check on type A. Such an assumption must be clearly stated. Reusable software should minimize dependence on the format of input not totally constrained or specified in the implementation language. For example, a function input of type STRING that is interpreted in two parts exhibits an "embedding of semantics" that makes reuse potentially difficult. If this is absolutely necessary it should be documented. However, it would be better if the function received two arguments in which the appropriate substrings were passed.

Finally, if it is necessary to pass data through a software module that will not be used directly in the module, but by a routine it calls, refrain from using this data in any decisions made in the module. Were this data to subsequently change, the assumption that it was "simply passed through" would be incorrect and the module would require an unexpected change.

14. Component is standardized in the areas of invoking, controlling, terminating its function [JONES84], error-handling, communication and structure [LANERGAN84].

Standardization is a key to understandability in software reuse. A template for writing software components or parts for reuse is desirable when people other than original code developers must deal directly with code. In the area of invoking a function, for example, standards can involve grouping of parameters with like modes together or placing all default parameters at the end of a parameter list. In an extreme case and for a particular application area, all functions of particular types will by convention have a common set of parameters. Data may be structured in a common way; control functions and function termination may be standardized as well. For example, one exit point per function may be desired or possibly one normal exit and a grouped set of "abnormal" exits may be specified.

Reusable software should handle errors in a predictable and effective way. It is important to the use or reuse software modules to be able to understand and predict how they will react to erroneous conditions. This predictability is important for reliability and confidence as well. When a software module is combined or grouped with others for reuse, it is even more important that errors be handled effectively for ease in debugging and isolating problems since, in general, the user of the reusable modules will not know much about their internal structures. The message "Error at line 473 of module x" is inadequate for two reasons. First, a more descriptive message relating to module behavior, its external view, data provided to the module, and so forth would be more appropriate. Second, if the real error was caused by code in another module, a traceback mechanism to the original condition that caused the error in the other module is important and useful but is not provided. As reliability and correctness of software is established, more and more inspection of data

passed from one module to the next will be the major method of debugging a composite program [BIENIAK85]. Error handling that supports this inspection is desired.

Reusable software should communicate via standard protocols. Software that communicates with other software (or hardware) in standard ways (i.e., ones already known and accepted by its users) facilitates reuse in that it will be easier to use, no learning will be involved, and its use will tend to be more error-free/successful.

All of the examples above are driving toward some common structure for code so that understandability and possibly data and control code interchangeability can make reuse possible.

15. Components should be written to exploit domain of applicability [NEIGHBORS84]; components should constitute the right abstraction and modularity for the application.

There are two points to be made here. First, developers should exploit language features that help promote modularity of code. A 2000 line piece of code is less usable/modifiable than 10 200-line ones from an understandability and possibly a recompilation standpoint. Large black-box pieces of code can be very useful for reuse. However, modifiability is enhanced if a "piece of the whole" can be changed and recompiled without effecting the whole. Second, for particular application areas, modularity may involve packaging the right functions together for use. Certain functions in an application area used together and in a particular way may be quite natural. This aspect of application domain should be exploited and not overlooked when writing reusable software. In this case, these functions should be "packaged" together and made easily accessible to potential users through use of appropriate language constructs.

Figure I2-1 relates each of the proposed characteristics to the

Characteristics of Reusable Software

metacharacteristics it promotes.

Characteristic	Metacharacteristics				
	1:findable	2:understand- able	3a: built	3b:fit	3c:conceptually sound
1		*		o	
2		o			*
3		o		*	
4	o	o	o		*
5		*		o	
6			*		
7		o	o		*
8		o		*	
9		*		o	
10		o			*
11	*	o	o		
12			o	*	
13		o	o	*	
14			o	*	
15		o			*

(KEY: * = major metcharacteristic supported;
o = other metcharacteristics supported)

Figure I2-1: Reusability Characteristics Realize the Metacharacteristics

I-3 THE ADA PROGRAMMING LANGUAGE AND REUSABILITY

Ada was developed for the United States Department of Defense (DoD) to help combat the crisis in software development. The DoD realized that any software development system for effective use DoD-wide would have to be founded on a standard programming language with constructs supporting good software engineering practices. Therefore, they developed requirements for such a language and, after deciding that no existing language met the requirements, initiated a language design competition that resulted in the Ada language. [BOOCH83a] gives a complete history of the Ada language.

I-3.1 ADA DESIGN GOALS

Although Ada can certainly be used in other application areas, it was designed for use in the development, execution, and maintenance of large, real-time, embedded systems. As stated in [DOD83], "Ada was designed with three overriding concerns: reliability and maintenance, programming as a human activity, and efficiency."

In the area of reliability and maintenance, Ada emphasizes readability vs. writability. Program variables must be explicitly declared; their types are invariant. Ada avoids error prone notations, such as allowing implicit declarations; its syntax is English like. Ada provides separate compilation of program units with the same consistency checking between separately compiled units as occurs within units. Separate compilation facilitates program development and maintenance.

Ada was designed for the human programmer. The language was kept as small as possible for its primary application domain. Its design was simplified where possible and an attempt was made to provide constructs that correspond to its expected users' intuitions. Also, because software systems are becoming more complex, and distributed development is becoming more common, Ada supports writing a program as a set of independently produced software components that are then assembled into a complete program. This separate development feature is a key to the Ada language design and helps keep the cost of reuse down; packages, private types, and generics support this feature.

In the area of efficiency "any proposed construct whose implementation was unclear or that required excessive machine resources was rejected" [DOD83].

I-3.2 ADA AND REUSABILITY

Ada's design goals all implicitly support code reusability. Central to this support is the fact that Ada is designed for the human programmer. The ability to independently write software components that are understandable to human readers and that can be assembled into a compilable program is important

The Ada Programming Language and Reusability

to reusability because it allows the production and use of libraries of software parts. It is "off-the-shelf" software parts, written to be reused, that will be reused.

Software reliability, maintainability and efficiency also contribute positively to reusability. Reliability contributes to user confidence in a software component, maintainability to understandability, and efficiency to feasibility for reuse.

The Ada language contains a rich set of features that support reusability. Some of the most important ones are:

- o packages,
- o separate compilation and checking across program units,
- o separate specifications and bodies for program units,
- o information hiding constructs (e.g., private types),
- o generics, and
- o strong typing.

In this guidebook, we find it helpful to distinguish between directly and indirectly reusable software and relate this to Ada. Directly reusable software is software that developers can search for and directly use. Examples are Ada package specifications. Indirectly reusable software is software that supports directly reusable software and provides it the environment, the ancillary definitions and data that it needs to perform correctly. Examples are Ada package bodies, subunits, and so forth. In the ideal case, indirectly reusable software is incorporated into a program under construction automatically by a software base management system. See Chapter II-10 for further information on directly and indirectly reusable software.

blank back page

SECTION II:

GUIDELINES FOR WRITING REUSABLE ADA SOFTWARE

II-1 INTRODUCTION

This Section presents specific guidelines on how to realize the reusability characteristics outlined in Section I in Ada code. It is meant to be a reference manual and is organized to follow chapters in the Ada Language Reference Manual [DOD83]. For example, Chapter II-10 deals with Program Structure and Compilation Issues. Each chapter contains a brief Ada language summary followed by guidelines for implementing the reusability characteristics and a cross reference between these guidelines and characteristics. In addition, following the statement of each guideline, we list the specific reusability characteristics it supports. In this Section, we distinguish between "users," that is, humans who reuse Ada source code and "clients," that is, other software that references Ada source code.

We suggest a particular reading order for those who want to familiarize themselves with Section II in preparation to use it as a reference manual. This order is as follows:

- o Chapter 1
- o Chapter 10
- o Chapter 7
- o Chapter 6
- o Chapter 9
- o Chapter 12
- o Chapter 8
- o Chapter 11
- o Chapter 3
- o Chapter 4
- o Chapter 5
- o Chapter 13
- o Chapter 14
- o Chapter 2.

In Version 1.0 of this guidebook, Chapters II-1, II-6, II-7, II-8, II-9, II-10, and II-12 are complete; Chapters II-2, II-3, II-4, II-5, II-11, II-13, and II-14 are not complete.

A Guidebook for Writing Reusable Source Code in Ada

II-2 LEXICAL ELEMENTS

II-2.1 ADA SUMMARY

II-2.2 GUIDELINES

II-2.2.1 LEXICAL ELEMENTS IN GENERAL

II-2.2.2 PRAGMAS

II-2.3 GUIDELINE/CHARACTERISTIC CROSS REFERENCE

II-3 DECLARATIONS AND TYPES

II-3.1 ADA SUMMARY

II-3.2 GUIDELINES

II-3.2.1 DECLARATIONS AND TYPES IN GENERAL

II-3.2.2 OBJECT AND NAMED NUMBER DECLARATIONS

II-3.2.3 TYPES AND SUBTYPES

II-3.2.3.1 SCALAR TYPES

-- discrete types (enumeration, integer); real types (floating, fixed point);
Discrete and real operations

II-3.2.3.2 Composite Types

--arrays, records

II-3.2.3.3 Access Types

II-3.2.4 Derived Types

II-3.2.5 Declarative Parts

-- pragma program_error

II-3.3 GUIDELINE/CHARACTERISTIC CROSS REFERENCE

II-4 NAMES AND EXPRESSIONS

II-4.1 ADA SUMMARY

II-4.2 GUIDELINES

II-4.2.1 NAMES

--indexed components, slices, selected components, attributes

II-4.2.2 Expressions

-- include literals

II-4.2.2.1 Operators

II-4.2.2.2 Type Conversions

II-4.2.2.3 Qualified Expressions

II-4.2.2.4 Allocators

II-4.3 Guideline/Characteristic Cross Reference

II-5 STATEMENTS

II-5.1 ADA SUMMARY

II-5.2 GUIDELINES

II-5.2.1 SIMPLE STATEMENTS

--assignment, exit, goto, delay, raise, procedure_call, return, entry, abort, code

II-5.2.2 Compound Statements

--if, loop, accept, case, block, select

II-5.3 GUIDELINE/CHARACTERISTIC CROSS REFERENCE

II-6 SUBPROGRAMS

II-6.1 ADA SUMMARY

Subprograms are a form of program unit in Ada. Subprograms are executed when called and are of two types: procedures and functions. A function call is an expression that returns a value; a procedure call is a statement. Subprograms may be written with separate declarations⁽¹⁾ which define the interface to their "function" and subprogram bodies which define implementation details for execution. Subprogram bodies always contain an entire subprogram specification and must repeat the specification given in a corresponding subprogram declaration. However, subprogram declarations are optional from an Ada language point of view. Subprograms have formal parameters with in, out and inout modes. Subprogram bodies may be expanded inline at the point of call for the purposes of code generation by use of a pragma. Named and default parameters may be used. Subprogram names may be overloaded; that is, multiple subprograms with identical names but different parameter and result type profiles may be declared and used. Function subprograms can be defined to overload arithmetic and other operators.

II-6.2 GUIDELINES

This chapter's guidelines for writing Ada subprograms support the reuse metacharacteristics described in Section I and most of the characteristics they imply. When discussing these guidelines, we use the term "interface" to mean a relationship between a subprogram and (1) its users, (2) client software, and (3) the environment (i.e., other software). Interfaces to users consist of subprogram behavior. Interfaces to client software consist of the things that software needs to be able to call and receive the subprogram results (i.e., subprogram name and return type, parameter names, types, modes). Interfaces to the environment consist of what the subprogram "imports" (e.g., globals, other subprograms) and what it "exports" (e.g., result values, access to some "protected" storage, say, in a package body).

II-6.2.1 Subprograms in General

G6-1: Separate subprogram declarations and bodies for ease of recompilation and modification.

(1) A subprogram declaration is a subprogram specification followed by a semicolon. A subprogram specification provides the type of subprogram (i.e., either procedure or function), parameter list, and return type for functions.

(Supports: 12, use with minimal modification; 7, separation of specification and body).

As is the case with other program units, we expect that a common way to reuse subprograms will be to write new bodies for existing specifications. A collection of subprogram bodies corresponding to one specification may exhibit different performance characteristics or implement different algorithms and are thus useful. Separating subprogram declarations from bodies will make substitution of new or changed bodies and subsequent compilation easier.

Chapter II-10 states that packages are the "unit of reusability." Library unit package specifications encapsulate directly reusable data and specifications of operations on data that characterize abstract objects; corresponding package bodies encapsulate indirectly reusable implementation details of operations and other data. Thus, subprogram declarations "belong" in package specifications and corresponding bodies "belong" in package bodies (except bodies for main programs, see below). Since in Chapter II-7, we prescribe a separation of library unit package specifications from their corresponding (secondary unit) bodies, the separation of subprogram declarations and bodies is "automatically" achieved.

G6-2: All reusable subprograms except a main program must be written within a library unit package.

(Supports: 4, object-oriented software).

The library unit package is the "unit of reusability;" Chapter II-10 explains the reasons for this decision. Thus, reusable subprograms must be in packages. These packages and their contents are the reusable software in a software repository; they are "glued" together by a main program which is invoked from the environment. If this gluing is automatic or easily specifiable in a very high-level-language, main programs do not have to be kept in a repository. It is the reusable parts that they glue together that are important. However, if a main program glues together a "system" which can be viewed as a potential component of other systems, then that program should be put in a package which will be catalogued as directly reusable software.

G6-3: Use subprogram declarations to specify interfaces to reusable objects. Use subprogram bodies to implement these interfaces and properties of the objects.

(Supports: 8, low coupling; 13, insulation from the environment; 12, minimal modifications necessary for reuse; 1, clear interface; 4, object-oriented);

The interfaces to reusable objects specified in subprogram declarations comprise a name, parameters of particular types and modes, and return types for functions. Subprogram bodies contain the executable code for reusable objects. This code performs useful work. We are saying that the use of both subprogram declarations and bodies is important. The only exception to this guideline is a main program callable from the environment rather than by other software. In this case, a body alone is sufficient. This guideline is

related to G7-2 prescribing that package specifications implement interfaces to object abstractions and their bodies implement specific details of these abstractions.

G6-4: Write subprogram interfaces at an appropriate abstract level.

(Supports: 2, appropriate level of interface).

In Chapter II-7 (see guidelines G7-2, G7-3) we recommend that the interfaces to instances of reusable objects implemented in packages be subprograms whose specifications are visible in the package specifications. Moreover, these subprograms should all be at an equivalent abstract level. Lower-level subprograms needed by the interface subprograms should not be included in the specification but be declared either entirely within a package body or in a separate package if they are themselves reusable. For example, the memory package in Chapter II-7, Example 7-a should not mix operations on bits with "read" and "store" operations on memory elements (i.e., words).

II-6.2.2 Subprogram Declarations

G6-5: First-level package-nested subprogram declarations should have a standard format including regions for purpose, parameter descriptions and associated documentation.

(Supports: 14, standard format; recommended information and organization supports: 11, documentation for findability; 9, readability; 1, interface clarity).

Figure II6-1 shows an example format for a first-level package-nested subprogram specification. Such a specification should contain at least this information, arranged in any reasonable way. While it is important that all information in the template be available, it is not crucial that this information be stored within source code. It is important, however, that the format in which information is kept be the same for every first-level subprogram unit in a particular library. In this guidebook, we include all information in the template.

```

-----
Procedure <procedure_name> <parameter_list>;
-- Purpose:
-- Explanation:
-- Keywords:
-- Parameter Description:
-- Associated Documentation:

Function <function_name> <parameter_list> return <typemark>;
-- Purpose:
-- Explanation:
-- Keywords:
-- Parameter Description:
-- Associated Documentation:
-----

```

Figure II6-1: Subprogram Declaration Templates

Figure II6-1 shows three fields, each with one or more subfields.

- o The Purpose field contains a brief summary of the purpose of the subprogram. The Explanation subfield includes enough information about the particular interface to the object abstraction implemented (see Chapter II-10) to determine if it is a potential candidate for reuse. This may also include a description of the typical client of this subprogram. The Keyword subfield contains keywords for use in cataloging the subprogram.
- o The Parameter description field contains a brief explanation of each subprogram parameter and return type for functions. Parameter and return value semantics in addition to those directly expressable in Ada should be noted.
- o The Associated Documentation field contains references to any documentation associated with the subprogram specification, for example, requirements and design information. This is only necessary if additions or refinements to the associated documentation field of the enclosing library unit or secondary unit package are necessary.

Note that no Revision History field appears in the subprogram specification template. This is because revision information will be contained in the Revision History field of the enclosing library unit or secondary unit package.

II-6.2.3 Subprogram Bodies

G6-6: Secondary unit (subunit) and first-level package body nested subprogram bodies should have a standard format, including regions for revision history, purpose, associated documentation, parameter description, assumptions/resources required, side effects, diagnostics, packages, data declarations, operations, and algorithmic code.

(Supports: 14, standard format; recommended information and organization supports: 11, documentation for findability; 9, readability; 1, interface clarity; 6, scaffolding).

By secondary unit, we mean, a subprogram subunit that corresponds to a subprogram body stub that appears in a secondary unit package body. These subunits are at the "library level," are candidates for reuse, and must be documented accordingly. Figure II6-2 shows an example format for secondary unit or first-level package-nested subprogram bodies. Such bodies should contain at least this information, arranged in any reasonable way.⁽¹⁾ While it is important that all information in the template be available, it is not crucial that this information be stored within source code. It is important, however, that the format in which information is kept be the same for every secondary unit or first-level subprogram body in a particular library. In this guidebook, we include all information in the template.

Figure II6-2 shows eleven fields, each with one or more subfields.

- o The Revision History field contains a revision number, date, name of modifier, and description of changes made. The initial code is the first revision. It is only necessary with secondary unit (i.e., subunit) subprograms. Otherwise, revision data is contained in the corresponding field in the enclosing secondary unit package.
- o The Purpose field has two subfields. The Explanation subfield contains a brief description of the purpose of the subprogram and specific implementation details that distinguish this subprogram body from others (e.g. performance characteristics, algorithms used, etc.) if they exist. The Keyword subfield contains keywords (in addition to those in the corresponding specification) used mainly to distinguish this subprogram body from other bodies.

(1) In extreme cases, this may require use of multiple fields and/or subfields with the same name. If this is necessary, we recommend using numbered indices to indicate successive fields/subfields (e.g., Subprograms (1), Subprograms (2), etc.).

```

[Seperate (parent_unit_name)] --for subunit subprograms only
Procedure <procedure_name> <parameter_list> is

  -- Revision History:
  -- Purpose:
  --   Explanation:
  --   Keywords:
  -- Associated Documentation:
  -- Parameter Description:
  -- Assumptions/Resources Required:
  -- Side Effects:
  -- Diagnostics:
  -- Packages:
  -- Data Declarations:
  --   Types:
  --   Objects:
  -- Operations:
  --   Subprograms:
  --   Tasks:
  -- Algorithm:
Begin ... End;

[Seperate (parent_unit_name)] --for subunit subprograms only
Function <function_name> <parameter_list> return <type_mark> is

  -- Revision History:
  -- Purpose:
  --   Explanation:
  --   Keywords:
  -- Associated Documentation:
  -- Parameter Description:
  -- Assumptions/Resources Required:
  -- Side Effects:
  -- Diagnostics:
  -- Packages:
  -- Data Declarations:
  --   Types:
  --   Objects:
  -- Operations:
  --   Subprograms:
  --   Tasks:
  -- Algorithm:
Begin ... End;

```

Figure II6-2: Subprogram Body Template

A Guidebook for Writing Reusable Source Code in Ada

- o The Associated Documentation field contains references to any documentation associated with the subprogram body, for example, requirements and design information. This is only necessary if additions or refinements to the associated documentation field of the enclosing secondary unit package are necessary.
- o The Parameter Description field contains a brief explanation of each subprogram parameter and the return type for functions, and clarifications of parameter semantics, especially those that are peculiar to this subprogram body and distinguish it from others that may exist. Just as multiple package bodies corresponding to one package specification may be useful (see Chapter II-7), so too may multiple subprogram bodies corresponding to one subprogram declaration within a library or secondary unit package. These subprogram bodies may implement different algorithms, have different performance characteristics, and so on. To implement these subprogram bodies and be consistent with our reusability guidelines, these subprogram bodies must be written as subunits corresponding to body stubs within secondary unit package bodies. While multiple source code versions of a subunit subprogram body may exist, only one compiled version may exist in an Ada library. This can be managed by a software base management system supporting reusability.
- o The Assumptions/Resources Required field contains any assumptions made by the subprogram body about conditions and/or resources required in its environment. Such assumptions may include host/target machine, operating system, physical devices (e.g. I/O), runtime utilities required or automatic interrupts affecting execution. This information need only be supplied if the current subprogram body (1) is not a subunit, is contained entirely within an enclosing secondary unit package body, and requires a refined list of resources as compared with the package body, or (2) is a subunit. Requirements for other modules are automatically documented in the context clauses associated with the current subprogram body (assuming it is a subunit) or the enclosing secondary unit package body.
- o The Side Effects field documents all side effects caused by the execution of any code in the subprogram body (e.g., changing of "global data"). Of course, we encourage the minimization of side effects.
- o The Diagnostics field contains an object-oriented description of error handling in this subprogram body and appropriate object-level exception declarations. We are promoting error handling at as high a level as possible, so that a user of a subprogram body can understand error messages with minimal familiarity with the details of the subprogram being reused. Use should be made of object level exceptions declared in an enclosing package scope. See Chapter II-7 and the package template diagnostics field. This planned approach to error handling is a form of scaffolding suggested in reusability characteristic #6.
- o The Packages field contains packages nested within the subprogram body.
- o The Data Declarations field contains two subfields, Types and Objects; they contain type and object declarations with associated documentation.

- o The Operations field contains two subfields, Subprograms and Tasks; they contain declarations and bodies of nested subprograms and tasks.
- o The Algorithm field contains the block associated with the subprogram body. The block should comprise commented code and exception handlers for exceptions raised during execution of this code and/or propagated exceptions raised by nested subprograms and task bodies.

G6-7: Write subprogram bodies to effectively handle interaction with/ effects on their environment.

(Supports: 3, minimal interference with the environment).

When one attempts to reuse code, one often discovers that a small modification in the environment is required to make that program perform correctly. However, after the program runs, the environment must be restored to enable environmental software to function properly. If this setting/restoration cannot be done, then the subject code cannot be reused. To manage this setting/restoration, a set of O-functions (object-changing functions) and V-functions (value-returning functions) is required. See Example 6-a.

Example 6-a:

```

-----
Procedure example is
  desired_XYZ, saved_XYZ: XYZ_state;
  procedure set_XYZ_state (XYZ : in XYZ_state);
  procedure get_XYZ_state (XYZ : out XYZ_state);
  .
  .
begin
  Get_XYZ_state (Saved_XYZ); -- save environmental state;
  Set_XYZ_state (Desired_XYZ); -- change environmental state;
  .
  .
  perform desired reusable action;
  .
  .
  Set_XYZ_state (Saved_XYZ); -- restore environmental state;
end example;
-----

```

Thus, as in Example 6-a, subprogram bodies that must alter an environmental parameter must save and restore it before and after execution of the main-line code in order to be reused with no lasting adverse effect on its environment.

A Guidebook for Writing Reusable Source Code in Ada

G6-8: Write subprogram bodies with one normal exit and a grouped set of abnormal exits via exception handlers.

(Supports: 14, standardization; 9, readability; 12, modifiability).

Function subprograms in Ada must contain at least one return statement; procedure subprograms may contain zero, one or more return statements. What we propose here is one normal (i.e., error-free) return or exit place and a grouped set of error exits for every subprogram. Among other things, this guideline implies that procedures should not contain any return statements and that their normal exit is at the end of their non-exception algorithmic code. This enhances readability in that a user knows where to look for subprogram exits; it enhances modifiability in that the exits that may need changing in order to reuse subprograms are in standard positions. See Example 6-b.

Note that this guideline implies an exception handler with an "others" alternative is a minimum requirement for all subprograms in which any exception (predefined or user defined) can be raised. This "others" alternative will (minimally) satisfy the grouped set of error exits suggested above.

Example 6-b:

```
Function A return B_type is
  c: B_type;
begin
  -- code to calculate c;

  return c;                      -- one normal exit;

  exception
    when others =>                -- grouped set of abnormal exists;
      raise object_level_exception;
      return c;
end A;
```

G6-9: Write subprogram bodies to pass results back to callers rather than use results to effect their function.

(Supports 5, no side-effects).

We write subprograms to produce some result. This result should be predictable. One way to make it predictable is to always pass results of execution back to client software callers and let them decide how to handle errors and/or how to proceed. This is especially important for function subprograms. Function calls appear in expressions that depend on calculated values they return. Calls to functions written "without side-effects" in expressions can make evaluation of expressions a standard, understandable process. Procedures written with out or in_out parameters to pass back

results of processing give client software callers an opportunity to use these results in their processing. For both procedures and functions, some kind of status information indicating whether any errors have occurred in processing is recommended (where errors are possible). Exceptions can be effectively used here. In general, following this guideline will help produce code that is composable and thus reusable in multiple environments.

II-6.2.4 Formal Parameter Modes

G6-10: Exploit formal parameter modes to clarify subprogram interface semantics.

(Supports: 1, interface clarity).

As [DOD83] states:

"A formal parameter of a subprogram has one of the three following modes:

- in The formal parameter is a constant and permits only reading of the value of the associated actual parameter.
- in out The formal parameter is a variable and permits both reading and updating of the value of the associated actual parameter.
- out The formal parameter is a variable and permits updating of the value of the associated actual parameter."

These modes should be used to constrain use of parameters to their intended function. Parameter modes should be appropriate to their function and parameter types should be appropriate to their expected values. Parameter modes in Ada provide a convenient way for programmers to express the semantics of their ideas in source code and let the language enforce these semantics. See Example 6-c.

Example 6-c:

```
-----
Subtype value is Integer range 0...100;

Procedure Read (X: out value); -- X can only be updated;
Procedure Write (X: in value); -- X can only be read;
Procedure Test_and_Set (X: in out value); -- X can be read
                                           and updated;
-----
```

II-6.2.5 Subprogram Calls, Default Parameters and Parameter Associations

G6-11: Use default parameters to generalize the context of a reusable subprogram; write complete subprogram specifications.

(Supports: 10, balance between generality and specificity).

This guideline says write a "complete" specification for every subprogram (i.e., provide parameters for its anticipated range of uses) even if one initially implements the code to use some but not all of the parameters or their values. Use default parameter expressions to give values to the "extra" parameters.

Whenever we write source code, we implicitly write it to exist in a particular environment. In that environment, there are objects, such as non-local data or procedures that the code uses, but are not explicitly specified. An attempt to reuse the code in another environment may fail because one of the implicit parameters is no longer the same. (Note that generic formal parameters would need to be used to specify subprogram parameter defaults. See Chapter II-12). Thus, when analyzing the requirements for a subprogram look at general needs as well as the needs at the current time. Always ask the question, "In what larger context could this be used?". Example 6-d shows an interface which precludes broad reuse, and better interfaces which facilitate reuse.

Example 6-d:

```
-----  
procedure Put (Item: in Character);      -- too restrictive, allows  
                                         -- output to only the "default  
                                         -- place"  
  
procedure Put (  
  Item: in Character;  
  In_Stream: in Stream := Standard_IO_Stream); --better declarations  
  
  - or -  
  
procedure Put (  
  Item: in Character;  
  In_Window: in Window := Standard_IO_Window);  
-----
```

Procedure Put, as defined by the first declaration in Example 6-d can only be used if its environment provides "global" default targets for its outputs or it is accompanied by a pair of definitions such as:

```
Set_Standard_IO_Stream (Standard_IO_Stream: in Stream);  
and  
Get_Standard_IO_Stream (Standard_IO_Stream: out Stream);
```

Note that it is NOT necessary for procedure Put to be able to handle output on multiple streams/windows for its initial implementation. However, successive versions of Put can include code to handle additional streams/windows. To elaborate, the first implementation of Put might be written to simply test the default parameter value (i.e., In_Stream), and reject any value but the default. A second implementation might handle some subset of streams while a final one might handle all.

No module using Procedure Put would need to be rewritten to use the increased actual function in subsequent implementations. New code can be written for the increased function without making the old obsolete, thus keeping it logically as well as physically reusable.

G6-12: Group all default parameters in subprogram parameter specifications at the end of the specifications.

(Supports: 14, minimize interference with environment; 12, ease of modification).

This guideline facilitates successive refinement of subprograms in three ways: (1) Adding to or modifying a subprogram's default parameters will not interfere with those client modules currently depending on it. Calls to a subprogram with default parameters need not supply values for these parameters. Therefore, addition of default parameters at the end of a parameter list or modification of existing ones will likely have no effect on subprogram calls from the environment since the calls can remain the same; (2) Changing the value of a default parameter is easier than changing code in the subprogram body proper (although recompilation will be required); (3) Changes to default parameters will all be done in one standard place. This will simply make changes easier, especially in long, formatted parameter lists.

G6-13: Use named parameter associations for calls on subprograms with more than three parameters or in any case for interface clarity.

(Supports: 1, interface clarity; 9, readability).

Calls to subprograms with more than three parameters can easily be confusing, especially if the parameters can take on values of similar types. Use of named parameter associations takes more effort when writing code than using positional associations. However, it clarifies subprogram interfaces from the client software or caller's point of view. It clearly states for reusers of the client software which "values" are intended for which parameters of called subprograms. Also, assuming descriptive formal parameter names are used in the specifications of subprograms, it provides an indication of parameter semantics at the points where these subprograms are called.

II-6.2.6 Overloading of Subprograms

G6-14: Minimize subprogram overloading.

A Guidebook for Writing Reusable Source Code in Ada

(Supports: 9, readability; 13, insulation from the environment).

Overloading allows the same subprogram name to be reused everywhere it may be meaningful or appropriate. However, readability and understandability decrease drastically when code contains several instances of an overloaded name, each instance bound to a different definition.

Overload resolution occurs at compile time. When an overloaded subprogram name is used, the compiler disambiguates the name depending on: (1) what definitions of the name are visible, (2) the types and numbers of parameters associated with the use of the name, and (3) for functions, the return type required by the context in which a function name is used. All three of these factors are determined by the context of use, that is, the name to type bindings in force when the overloaded subprogram name is used. Changes in the context of use of an overloaded subprogram that lead to erroneous and/or unintended results may not be caught by an Ada compiler, while the same context change will likely flush out erroneous effects on non-overloaded entities.

Example 6-e:

```
LHS,x: Integer;  
y,z: Float;  
  
Function Example (A, B: Float) Return Float is ... --(1)  
Function Example (A, B: Float) Return Integer is... --(2)  
Function Example (A: Integer; B: Float) Return Float is... --(3)  
.  
.  
.  
LHS := Example(Example(X,Y), Z);
```

In Example 6-e, the innermost call to function Example in the expression for LHS resolves to overloaded definition (3) and the outermost to overloaded definition (2). This is not obvious however and at a minimum, lowers program readability. Definition (1) of Example is not used. To show how volatile a change in context can be, assume the user provided a context with y, and z both of type FLOAT but "inadvertantly provided x of type FLOAT rather than INTEGER; then the inner Example function call would resolve to overloaded definition (1) while resolution of the outermost call would remain unchanged. While this was not what the programmer or reuser intended, no compiler error will result since the compiler can find a valid albeit incorrect resolution for the expression's components. If overloaded Example functions were not used, as in Example 6-e, name resolution would fail since a definition appropriate to each function call would not be found. (See Example 6-f.) The code of Example 6-f is more "insulated" from its context or environmental namespace or at least insulated from a complicated environment because it admits only one interpretation.

Example 6-f:

```
LHS : Integer;
x,y,z: FLOAT;
```

```
Function Example_1 (A, B: Float) Return Float is...
Function Example_2 (A, B: Float) Return Integer is...
Function Example_3 (A: Integer; B: Float) Return Float is...
```

```
LHS := Example_2 (Example_3 (x,y), z); -- compiler will find no
                                         definition for Example_3
                                         with two Float parameters.
```

The potential complications of overloading described above will most likely occur if reuse takes the form of combining code fragments together rather than combining entire modules through use of "with" but not "use" context clauses and previously compiled library units. Minimizing use of use context clauses will provide some help to insulate subprogram names from context changes since references to all names declared within named packages must be fully qualified. Thus, a change in context implemented by a change in packages referenced in a context clause will require explicit changes in subprogram calls and no unintentional results as described above with respect to Example 6-e.

If a "new" context for the assignment statement and declarations in Example 6-e contained an exact duplicate declaration rather than a legal overloaded declaration of one or more of the Example functions, a compiler would catch one or more illegal homographs.

II-6.3 GUIDELINE/CHARACTERISTIC CROSS REFERENCE

Figure II6-3 provides a cross reference between the reusability guidelines presented in this chapter and the characteristics in Chapter I-2.

A Guidebook for Writing Reusable Source Code in Ada

	Reusability Characteristic														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Guideline															
G6-1							X					X			
G6-2				X											
G6-3	X			X				X				X	X		
G6-4		X													
G6-5	X								X		X			X	
G6-6	X					X			X		X			X	
G6-7			X												
G6-8									X			X		X	
G6-9					X										
G6-10	X														
G6-11										X					
G6-12												X		X	
G6-13	X								X						
G6-14									X				X		

Figure II6-3: Guideline/Characteristic Cross Reference

II-7 PACKAGES

II-7.1 ADA SUMMARY

Packages are one form of Ada program unit. They allow conceptually related entities such as type and object declarations and callable operations to be grouped together. Packages have specifications and bodies. Package specifications contain a public part that is visible (that is, referencable) by client software and a private part that is not. Package bodies contain implementation details, e.g., data, subprograms and tasks; their contents are not visible to client software. A common use of packages is to provide data and specifications for operations (subprograms or tasks) on this data in package specifications and corresponding bodies for these operations in associated package bodies.

This distinction between public and private parts of a package can be used to separate logical properties of data that should be used by client software from properties that should be used only by the package itself. Operations on private types declared within a package are limited to implicit operations⁽¹⁾ and explicit operations declared within the package specification.

II-7.2 GUIDELINES

The following guidelines for writing Ada packages support mainly the feasibility for reuse metacharacteristic and give some support to findability and understandability.

Packages have been selected as the encapsulation mechanism for reusable software. This is their role in our reusability scheme; we deal with that role in Chapter II-10. Here we present guidelines for creating packages that are reusable because of the nature of their design and code.

When considering what information potential users of a package need to convince themselves that the package provides a desired function, an important point becomes apparent: There are details a potential user may need to see that are irrelevant to client software (e.g., the structure of a particular type or object). While we hope that eventually software reuse will not involve examining those details, we feel that it is an important aspect to consider today. Providing all implementation details in the package specification would defeat Ada's information hiding benefits. Those details will

(1) The implicit operations are: assignment (unless the type is limited), selected components for discriminant selection, membership tests, qualification, explicit conversions, attribute operations, and unless the type is limited, comparison for equality and inequality.

appear in the private parts of package specifications and in package bodies. Because those details may be examined by humans, private parts of package specifications and package bodies must to some extent be written according to reusability guidelines. Obviously, the visible part of package specifications must obey reusability guidelines.

Details which users need to know and which cannot appear in package specifications include, for example, information that distinguishes one package body from another, where both correspond to the same package specification. This information must reside in the bodies. To discover this information, users must examine the bodies directly.

Therefore, different guidelines in this chapter support each of two sometimes conflicting goals: (1) using Ada language constructs to facilitate reuse by client software (e.g., information hiding) and (2) using Ada language constructs to facilitate reuse by a user. In the future, automatic extraction of information about package specifications and bodies can make this conflict disappear.

II-7.2.1 Package Structure

G7-1: Write library unit package specifications and bodies in separate files for ease of recompilation and modification.

(Supports: 12, use with minimal modification; 7, separation of specification from body).

We expect that a common way to reuse software will be to write new package bodies for existing specifications. A collection of package bodies corresponding to one specification may exhibit different performance characteristics or implement different algorithms and are thus useful. Using separate files for package specifications and their bodies will make substitution of new or changed bodies and subsequent recompilation easier.

G7-2: Use package specifications to specify the interface to object abstractions; use package bodies to encapsulate implementation-specific details of these abstractions not needed by client software.

(Supports: 8, low coupling; 13, insulation of a component from its environment; 12, use with minimal modification; 1, clear interface; 4, object-oriented);

Simply stated, decide what object abstraction a package should implement, decide what the interface to this abstraction should be, and implement these as visible specifications for operations on data in the public part of a package specification. Decide what the implementation structure of the abstraction should be and implement this and all other details in the private part of the package specification and a corresponding package body. This separation benefits the package itself and its environment. The less "connec-

tion" a package has with the outside world (e.g., the smaller the visible part of a package specification), the lower its coupling with other modules. Once modules in a package's environment begin to depend on particular visible entities that really should have been hidden, the package becomes less and less insulated from its environment.

Assume the declaration of a subprogram or task serving as an interface to a reusable object is contained in a package specification and the corresponding body is contained in a corresponding package body. When the subprogram or task's data structures and operations but not its interface need modification, only the package body will need to be recompiled.

There are two strategies for providing abstractions as reusable objects [BOOCH85]:

- o Provide the basis for multiple "public" reusable objects with common operations on the objects. Do this by writing type declarations and specifications for operations on data of these types in package specifications and implementations of the operations in corresponding package bodies. The object abstraction can then be reused by client software (multiple times) by declaring variables (external) to the package and using the operations provided by the package to manipulate these variables.
- o Provide single, sharable, "private" reusable objects and operations on these objects. Do this by encapsulating types of reusable objects in package bodies. This limits client software from declaring and using multiple instances of the reusable objects since their types are hidden. Provide specifications for operations on reusable objects in package specifications. Provide variable declarations for the reusable objects and implementations of operations on the objects in corresponding package bodies. These operations must be parameterless in the case where the types of the reusable objects are not "composite." These operations may contain parameters if the types of the reusable objects are "composite," and "atomic" public types from which these types are constructed are declared in package specifications. Client software can only reuse the specific instances of object abstractions contained in these packages. This software can only indirectly access the variables implementing reusable objects through interfaces provided by visible subprograms specified in the package specifications. Note: The only way to obtain multiple instances of these object abstractions would be to use generics. See Chapter II-12.

The first strategy is fairly straight-forward and we will not provide an example of it. However, we do provide Example 7-a as a sample of the second strategy. Consider a memory package that forms part of the simulation of a computer. One memory is simulated and access to its contents is provided through special routines. "Atomic" types for memory_location and memory_elements are declared in the package specification, along with operations "read" and "store" specified with parameter and/or return values of these types. The actual object (variable) that implements memory, main_memory, is of type memory_type which is built up from these atomic types. Both the reusable object (main_memory) and its associated type declaration (memory_type) reside in a corresponding package body hidden from direct use by

client software. Main memory can only be indirectly referenced with "read" and "store" operations. Direct reference to memory could have been achieved by placing the declaration for memory_type and the variable declaration for main_memory in the visible part of the memory package specification. However, this would have created an open-ended, unclear, unpredictable interface to memory. With direct access to memory, any client software can reference and possibly change any memory location in any way it wants. Specifying indirect access to memory as in Example 7-a, provides a standard and clear interface. The "read" and "store" subprograms which make up this interface could possibly implement memory access synchronization. This synchronization, while important, would be hidden from client software due to its implementation in the subprogram bodies for "read" and "store" in the memory package body.

Example 7-a:

```
Package Memory is
  Type memory_location is range 0..3000;
  Type memory_element is array (INTEGER range 0..31) of BOOLEAN;
  .
  .
  .
  Function Read (X: memory_location) return memory_element;
  Procedure Store (X: memory_location; Y: memory_element);
End Memory;

Package Body Memory is
  Type memory_type is array (memory_location) of memory_element;
  Main_memory: memory_type;
  Function Read...end Read;
  Procedure Store...end Store;
End Memory;
```

We recommend using separate compilation, body stubs, and subunits to achieve modifiability and modularity. While it is important to package-up all the data and operations associated with a reusable object, body stubs in package bodies can be used to achieve the required modularity. See Chapter II-10, for further details.

G7-3: Packages should implement interfaces to reusable objects at a consistent abstract level.

(Supports: 2, appropriate abstract level;)

As stated above, we recommend that the interfaces to instances of reusable objects implemented in packages be subprograms and/or tasks whose specifications are visible in the package specifications. Moreover, these subprograms and tasks should all be at an equivalent abstract level. Lower-level subprograms, for example, needed by the interface subprograms or tasks should

not be included in the specification but be declared either entirely within a package body or in a separate package if they are themselves reusable. For example, the memory package in Example 7-a should not mix operations on bits with its "read" and "store" operations on memory_elements (i.e., words). An excellent example of layered abstractions is provided by the Graphics Kernal System (GKS) [HARRIS85].

II-7.2.1.1 Package Specifications and Declarations

G7-4: Library unit package specifications should have a standard format, including various regions for revision history, purpose, associated documentation, diagnostics, packages, data declarations, operations, and private types.

(Supports: 14, standard format; recommended information and organization supports: 11, documentation for findability; 9, readability; 6, scaffolding; 1, interface clarity).

Figure II7-1 shows an example format for a library unit package specification. A library unit specification should contain at least this information, arranged in any reasonable way.⁽¹⁾ While it is important that all information in the template be available, it is not crucial that this information be stored within source code. It is important, however, that the format in which information is kept be the same for every unit in a particular library. In this guidebook, we include all information in the template.

(1) In extreme cases, this may require use of multiple fields and/or subfields with the same name. If this is necessary, we recommend using numbered indices to indicate successive fields/subfields (e.g., Subprograms (1), Subprograms (2), etc.).

```
Package <package_name> is

    -- Revision History:
    -- Purpose:
    --   Explanation:
    --   Keywords:
    -- Associated Documentation:
    -- Diagnostics:
    -- Packages:
    -- Data Declarations:
    --   Types:
    --   Objects:
    -- Operations:
    --   Subprograms:
    --   Tasks:
    -- Private:

End <package_name>;
```

Figure II7-1: Package Specification Template

Figure II7-1 shows eight fields, each with one or more subfields.

- o The Revision History field contains a revision number, date, name of modifier, and description of changes made. The initial code is the first revision.
- o The Purpose field contains a brief summary of the purpose of the package. The Explanation subfield includes enough information about the object abstraction implemented (see Chapter II-10) to determine if it is a potential candidate for reuse. This may also include a description of the typical client of this package. The Keyword subfield contains keywords used to catalog the unit.
- o The Associated Documentation field contains references to any documentation associated with the package specification, for example, requirements and design information.
- o The Diagnostics field contains object-level exception declarations which explain errors at the object-abstraction level rather than at the level of the code that implements the abstraction. We are promoting high-level error handling so that a user of a package can understand errors without knowing the package body details. These object-level exceptions will be used by clients of the package and in corresponding package bodies. This approach to error handling is a form of the scaffolding promoted in reusability characteristic #6.
- o The Packages field contains specifications of nested packages.

- o The Data Declarations field contains two subfields, Types and Objects; they contain type and object declarations with associated documentation.
- o The Operations field contains two fields, Subprograms and Tasks; they contain specifications of the subprograms and tasks encapsulated in this package.
- o The Private field contains full type definitions for corresponding partial definitions given in the Types subfield. This is required by the Ada language.

For languages other than Ada we would suggest a dependency field to list other software modules required by the current one. In Ada this is automatically documented in context clauses on package specifications.

Specific resources required, such as host operating system, should be specified in the corresponding package body or bodies.

As discussed in Chapter II-10, we recommend that only first-level nested non-package entities in library unit package specifications form the basis for catalogued directly reusable objects/software.

II-7.2.1.2 Package Bodies

G7-5: Secondary unit package bodies should have a standard format including regions for revision history, purpose, associated documentation, assumptions/resources required, side effects, diagnostics, packages, data declarations, operations and initialization code.

(Supports: 14, standard format; recommended information and organization supports: 11, documentation for findability; 9, readability; 6, scaffolding; 1, interface clarity).

Figure II7-2 shows an example format for a secondary unit package body. A secondary unit body should contain at least this information, arranged in any reasonable way.⁽¹⁾ While it is important that all information in the template be available, it is not crucial that this information be stored within source code. It is important, however, that the format in which information is kept be the same for every unit in a particular library. In this guidebook, we include all information in the template.

(1) In extreme cases, this may require use of multiple fields and/or subfields with the same name. If this is necessary, we recommend using numbered indices to indicate successive fields/subfields (e.g., Subprograms (1), Subprograms (2), etc.).

Figure II7-2 shows ten fields, each with one or more subfields. The Revision History, Associated Documentation, and Data Declaration fields have the same purpose as their counterparts in the package specification template above.

- o The Purpose field has two subfields. The Explanation subfield contains a brief description of the purpose of the package and specific implementation details that distinguish this package body from others (e.g. performance characteristics, algorithms used, etc.) if they exist. The Keyword subfield contains keywords (in addition to those in the corresponding specification) used mainly to distinguish this package body from other bodies.

```
Package body <package_name> is

  -- Revision History:
  -- Purpose:
  --   Explanation:
  --   Keywords:
  -- Associated Documentation:
  -- Assumptions/Resources Required:
  -- Side Effects:
  -- Diagnostics:
  -- Packages:
  -- Data Declarations:
  --   Types:
  --   Objects:
  -- Operations:
  --   Subprograms:
  --   Tasks:
  -- Initialization:

  [begin

  [exception]]

end [<package_name>];
```

Figure II7-2: Package Body Template

- o The Assumptions/Resources Required field contains any assumptions made by the package body about conditions and/or resources required in its environment. Such assumptions may include host/target machine, operating system, physical devices (e.g. I/O), runtime utilities required or automatic interrupts affecting execution. Requirements for "withed" modules are automatically documented in the context clauses on this package body and its corresponding package specification.

- o The Side Effects field documents all side effects caused by the execution of any code in the package body (e.g., changing of "global data"). Of course, we encourage the minimization of side effects.
- o The Diagnostics field contains an object-oriented description of error handling in this package body and appropriate object-level exception declarations supplementing those found in the corresponding package specification. We are promoting high-level error handling so that a user of a package can understand errors without knowing the package body details. These object-level exceptions will be used by clients of the package and in corresponding package bodies. This approach to error handling is a form of the scaffolding promoted in reusability characteristic #6.
- o The Packages field contains bodies of visible packages whose specifications appeared above in the corresponding package specification. It can also contain declarations of additional packages.
- o The Operations field contains two subfields, Subprograms and Tasks; they contain the bodies of visible subprograms and tasks whose specifications appeared above in the corresponding package specification. They can also contain declarations of additional subprograms or tasks.
- o The Initialization field contains the block associated with package bodies. This block contains commented data initialization code and exception handlers for exceptions raised during execution of this code and/or propagated exceptions raised by nested subprograms and/or task bodies.

II-7.2.2 Private Type and Deferred Constant Declarations

G7-6: Use private or limited private types and the private part of package specifications to restrict client software's view of data and operations on that data.

(Supports: 8, low coupling; 13, insulation from the environment).

For the same reasons given in the discussion of guideline G7-2, using private/limited private types and the private part of package specifications can minimize module coupling and insulate modules from their environment. Private types allow strict specification of client software's view of a package. As stated in [DOD83],

"The declaration of a type as a private type in the visible part of a package serves to separate the characteristics that can be used directly by outside program units (that is, the logical properties) from other characteristics whose direct use is confined to the package (the details of the definition of the type itself)."

The structural characteristics of private types are invisible outside the package specification in which they are declared. Private types have associated implicit operations (such as assignment, comparison for equality

and inequality) as well as explicit operations defined by subprograms with a parameter or result type of the private type. These explicit operations can redefine and hide implicit ones. Additional implicit operations declared by the corresponding full type declaration are also available inside of the package itself (e.g., arithmetic operators). Limited private types are even more restrictive. Assignment, equality and inequality operators are not implicitly declared.

Example 7-b taken from [DOD83] illustrates the power of private types.

Example 7-b:

"

```

package I_O_PACKAGE is
  type FILE_NAME is limited private;

  procedure OPEN (F : in out FILE_NAME);
  procedure CLOSE (F : in out FILE_NAME);
  procedure READ (F : in FILE_NAME; ITEM : out INTEGER);
  procedure WRITE (F : in FILE_NAME; ITEM : in INTEGER);
private
  type FILE_NAME is
    record
      INTERNAL_NAME : INTEGER := 0;
    end record;
end I_O_PACKAGE;

package body I_O_PACKAGE is
  LIMIT : constant := 200;
  type FILE_DESCRIPTOR is record ... end record;
  DIRECTORY : array (1 ... LIMIT) of FILE_DESCRIPTOR;
  ....
  procedure OPEN (F : in out FILE_NAME) is ... end;
  procedure CLOSE (F : in out FILE_NAME) is ... end;
  procedure READ (F : in FILE_NAME; ITEM : out INTEGER) is...end;
  procedure WRITE (F : in FILE_NAME; ITEM : in INTEGER) is...end;
begin
  ...
end I_O_PACKAGE; "
```

In Example 7-b, File_Name is a limited private type. It has only four operations, and these are explicitly given in the package specification: open, close, read and write. Other operations such as assignment and comparison for equality are not allowed. In this case, the writer of the package completely restricts client software's view of file structures and the operations that can be performed on them by exploiting Ada language features that allow accurate specification of how the client can use the code.

II-7.2.3 Additional Considerations

The Ada Language Reference Manual [DOD83] contains some rather detailed rules for private types, their implicit and explicit operations and deferred constant declarations. We advise their user to read the manual carefully when incorporating these constructs into a package.

In guidelines G7-4 ad G7-5 we advocate use of standard templates for package specifications and bodies. One field in the templates is the Revision History field. In it is a history of the changes made to a package since its creation. Analogous to this to further support reusability, we suggest a "reusability history" be kept on reusable packages. This history can help us to study reusability and suggest ways to reuse software to other "potential reusers." Automated tools to collect/retrieve this information and a software base in which to store it are desirable.

II-7.3 GUIDELINE/CHARACTERISTIC CROSS REFERENCE

Figure II7-3 provides a cross reference between the reusability guidelines presented in this chapter and the characteristics in Chapter I-2.

	Reusability Characteristic														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Guideline															
G7-1							X					X			
G7-2	X			X				X				X	X		
G7-3		X													
G7-4	X					X			X		X				X
G7-5	X					X			X		X				X
G7-6								X					X		

Figure II7-3: Guideline/Characteristic Cross Reference

II-8 VISIBILITY RULES

II-8.1 ADA SUMMARY

Ada visibility rules define where in program text identifiers, other symbols, and operations are visible. Use clauses allow direct visibility of declarations appearing in visible portions of package specifications. Renaming declarations can be used as shorthand notations and to resolve name conflicts. Package Standard is a predefined package containing standard types (i.e., BOOLEAN, INTEGER, ...) and forms a declarative region that effectively encloses every library unit [DOD83].

II-8.2 GUIDELINES

This chapter's guidelines for use of Ada visibility rules support about one-third of the reusability characteristics described in Section I. They center around writing software that requires minimal modification for reuse and minimizes interference with/maximizes insulation from its environment.

II-8.2.1 Use Clauses

G8-1: Do not use "use" context clauses.

(Supports: 12, minimal modification of software; 3, minimal interference with the environment).

The "use" context clause can lead to ambiguous and/or unexpected results because of Ada visibility rules. For example, in Example 8-a, the developer of package body B expected the reference to E_type to resolve to E_Type from package C. Package C was "withed" and "used" to do this. However, Ada's visibility rules caused the E_Type from the package specification for A to be used. This happened because (1) package A also declared a type named E_Type, (2) package A was "withed" and "used" in context clauses on the package specification for B, and (3) the E_Type declared in package A hides E_Type from package C.

Example 8-a:

```

-----
--assume packages A and C exist and both declare type E_type--

With A; Use A;
Package B is
.
.
.
end B;

With C; Use C;
Package Body B is
  Procedure D is
    For I in E_type'first...E_type'last Do
      LOOP
        .
        -- E_type from Package C expected; E_type from package A used.
        .
      end LOOP;
    end D;
  end B;

```

Identical identifiers and subprograms with the same parameter and result type profiles (i.e., homographs) will not be made visible by use of two "use" clauses for the packages that contain them, leading to unexpected results, as in Example 8-b.

Example 8-b:

```

-----
--Assume both packages A and B exist and declare type E_Type.

      (i)                      (ii)
With A; Use A;                With A,B; Use A,B;
Package C is                  Package C is
  x : E_type;                  x : E_type;  --E_type will be unresolvable
                                --unless fully qualified as
                                -- A.E_type or B.E_Type.
.
.
.
end C;                          end C;

```

In Example 8-b, assume package C is "catalogued" for reuse. In (i), the E_Type defined in package A is visible and used in package specification C. However, assume that in order to reuse package specification C, a reference to package B in a context clause as in (ii) is needed. If packages A and B both declare type E_type, Ada language rules state that neither E_Type from package A nor E_Type from package B is visible in package C because any reference to E_Type would be ambiguous. Thus the reference to E_type will be unresolvable unless it is fully qualified; that is, written as either A.E_type or B.E_type. A change to package specification C will have to be made, one that would have been unnecessary if no use clauses were used originally and fully qualified names were used within C. Package specification C, written with a use clause for A, interfered with the environment for reuse for C. Again we see that "use" clauses can lead to confusion and necessitate changes unforeseen during original development.

II-8.2.2 Renaming Declarations

G8-2: Use renaming declarations to resolve name conflicts with the environment.

(Supports: 9, readability; 13, insulation with the environment).

Renaming declarations can improve readability in that expanded names for packages, for example, can be "replaced" with simpler names. Note that after renaming declarations are used, both the old and new names are visible. They can be used to resolve name conflicts that arise due to the use of use clauses by providing distinct and simpler names for identically-named constructs in two or more packages. Without use clauses, these names would have to be fully expanded to be visible. However, we have already recommended that use clauses not be used (see above).

In addition, renaming declarations are a language-supported way of specifying "services" or "entities" required from the environment, as in Example 8-c. In this context renaming enforces nothing, but if it is used consistently, it is a form of documentation.

 Example 8-c:

```

Package Stack is
  Procedure Push ...;
  Procedure Pop ...;
  Procedure Top ...;
end Stack;

With Stack;
Package Menu_Manager is
  Procedure Enter_New_Menu renames Stack.Push;
  Procedure Return_To_Previous_Menu renames Stack.Pop;
end Menu_Manager;
  
```

Example 8-c shows a somewhat rudimentary form of insulation from the environment in that "local" redeclarations of "services" provided by the environment are given within a reusable software module. If the environment changes, it is obvious what its effect will be on the reusable module. Contrast this with an "undocumented", complex interaction of a "reusable" module with its environment and the effect of an environmental change. In the example, package Menu_Manager requires services from package Stack. These services are explicitly called out in the renaming declarations. If package Stack were changed, it would be obvious if the change affected package Menu_Manager, i.e., a change to Stack.Push or Stack.Pop would affect Menu_Manager while a change to Stack.Top would not. (We distinguish between simply having to recompile Menu_Manager if the specification for Stack changes as opposed to having to change something in Menu_Manager in addition to recompiling it).

G8-3: Use renaming declarations to facilitate modifying reusable software to represent new object abstractions.

(Supports: 12, modifiability; 4, object-oriented software).

We assume that subprograms with the right functionality but wrong names and parameter names to fit an object abstraction will be available fairly often. The same is true for the names of exceptions, variables, and packages. To illustrate this guideline, we will use subprogram renaming.

Renaming subprograms allows a user to introduce new parameter names and default expressions that differ from those in the renamed subprogram. Renaming minimizes the work necessary to reuse a subprogram since no recompilation is required. Thus, a subprogram can be reused without recompilation by using a renaming declaration to (1) change a default expression for a parameter of mode in, or, (2) change the subprogram name, parameter names, and/or possibly default parameter expressions. Option 2 facilitates translation between object abstractions; this is important to reusability "aesthetics." See Example 8-d. (Example 8-c also illustrates this guideline, but to a lesser extent).

Example 8-d:

```
Package Ballistic_Missile_Menu_Manager is
  Type target_type is range 0..50;
  Function Create_Ballistic_Missile_Menu
    (target_selection: target_type
... )...;
  Procedure Print_Ballistic_Missile_Menu ...;
  .
  .
end Ballistic_Missile_Menu_Manager;

With Ballistic_Missile_Menu_Manager;
Package Financial_Operations is
  Subtype operation_type is
    Ballistic_Missile_Menu_Manager.target_type range 0..10;
  Function Create_Financial_Ops_Menu (ops_selection: operation_type
... )...
    renames Ballistic_Missile_Menu_Manager.Create_Ballistic_Missile_Menu;
  Procedure Print_Financial_Ops_Menu
    renames Ballistic_Missile_Menu_Manager.Print_Ballistic_Missile_Menu;
  .
  .
end Financial_Operations;
```

II-8.2.3 Package Standard

G8-4: Do not hide package standard.

(Supports: 3, minimal interference with the environment; 9, readability).

Ada provides package standard as the environment for all programs. Using the identifier "standard" for a user_defined package or other declaration will hide package standard. This will interfere with its use and be confusing to the user of the software. Hiding of package standard may be acceptable in a software module's initial context but when its extended context for reusability is considered, hiding package standard will likely cause more harm than good. See Example 8-e.

Example 8-e:

```

Package A is
  Type standard is ...;
  x : standard;
  Function "+" (a1:integer; a2:integer) return integer;
  Procedure abc (a1:integer);
end A;

Package body A is
  Function "+" (a1:integer; a2:integer) return integer is ...end "+";
  Procedure abc (a1:integer) is
    y:integer;
    begin --abc
      -----
      | y:= standard."+"(a1,50); | --code added to reuse package A; reference
      -----                  --is illegal since standard refers to type
                                --not predefined package and "+" overloaded.

    end abc;
  end A;
  
```

In Example 8-e, assume a call to package standard's predefined binary adding operator for integers (i.e., "+") must be added to procedure body abc in order to reuse it. This function cannot be used, however, since it is effectively hidden. Hiding occurs because (1) the function "+" in package A hides package standard's operator "+" on integers so there is no direct visibility to the standard operator and (2) the type "standard" in package A hides the identifier "standard" referring to the predefined package and so there is no indirect visibility to the standard operator either. A reference to standard."+" is illegal.

II-8.3 GUIDELINE/CHARACTERISTIC CROSS REFERENCE

Figure II8-1 provides a cross reference between the reusability guidelines presented in this chapter and the characteristics in Chapter I-2.

A Guidebook for Writing Reusable Source Code in Ada

	Reusability Characteristic														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Guideline															
G8-1			X									X			
G8-2									X				X		
G8-3				X								X			
G8-4			X						X						

Figure II8-1: Guideline/Characteristic Cross Reference

II-9 TASKS

II-9.1 ADA SUMMARY

Tasks are a form of program unit in Ada. Like subprograms, they have declarations and bodies. Unlike subprograms, they (1) can be executed in parallel with other tasks and/or subprograms and (2) cannot be a compilation unit by themselves, but must be nested within a subprogram or package. Tasks may have entries which can be called from other tasks. Accept statements specify actions to be performed when an entry call is accepted. Calling and called tasks are synchronized by a rendezvous. Entries may have parameters which allow communication of values between calling and called tasks. Ada provides a set of statements for affecting task interaction. In addition to entry call and accept statements, these statements include delay, select, and abort statements. Task types and task objects can be defined. Priorities may be assigned to affect task execution (assuming the pragma PRIORITY is supported by the Ada implementation in use). Finally, shared variables may be declared (assuming the pragma SHARED is supported by the Ada implementation in use).

II-9.2 GUIDELINES

This chapter's guidelines for writing Ada tasks support the reuse metacharacteristics described in Section I and most of the characteristics they imply. Tasks can be considered as procedure subprograms that can execute in parallel with other tasks. Therefore, most of the reusability guidelines for subprograms contained in Chapter II-6 apply to tasks as well. Corresponding guidelines are enumerated below. Explanations of them are minimal and include a reference to Chapter II-6.

II-9.2.1 Tasks in General

G9-1: Separate task declarations and bodies for ease of recompilation and modification.

(Supports: 12, use with minimal modification; 7, separation of specification from body).

See Guideline G6-1.

G9-2: Use task declarations to specify interfaces to reusable objects. Use task bodies to implement these interfaces and properties of the objects.

(Supports: 8, low coupling; 13, insulation from the environment; 12, minimal modifications for reuse; 1, interface clarity; 4, object-oriented).

See guideline G6-3. For tasks, interfaces are concerned not only with parameter passing but also with synchronization. Later in this chapter we will discuss statements affecting this synchronization and their effect on reusability. Also, while subprograms can optionally have a separate declaration and body, tasks must have both declarations and bodies. Thus reusability characteristics #8, #13 and #12 are satisfied somewhat automatically. Characteristic #1 can best be achieved by specifying names and parameters for entries in a clear, well-documented manner. Characteristic 4 is satisfied by treating tasks as interfaces to reusable objects.

G9-3: Write task interfaces at an appropriate abstract level.

(Supports: 2, appropriate interface level).

See guideline G6-4.

II-9.2.2 Task Declarations

G9-4: First-level package-nested task declarations should have a standard format including regions for purpose, entry descriptions, representation clause descriptions, and associated documentation.

(Supports: 14, standard format; recommended information and organization supports: 11, documentation for findability; 9, readability; 1, interface clarity).

Figure II9-1 shows an example format for a task specification. A first-level task specification should contain at least this information, arranged in any reasonable way. While it is important that all information in the template be available, it is not crucial that this information be stored within source code. It is important, however, that the format in which information is kept be the same for every first-level task unit in a particular library. In this guidebook, we include all information in the template.

```

Task [type] <task_simple_name> [is
    -- Purpose:
    --   Explanation:
    --   Keywords:
    -- Entries:
    -- Representation Clauses:
    -- Associated Documentation:

end [<task_simple_name>]];

--Note: Square brackets indicate optional syntax.

```

Figure II9-1: Task Declaration Template

Figure II9-1 shows four fields, each with one or more subfields. See the discussion for guideline G6-5 for a description of the Purpose and Associated Documentation fields. Unlike subprograms, no Revision History field appears in the template because tasks cannot be library/compilation units.

- o The Entries field contains task entry declarations along with an explanation of their purpose, description of their parameters, and conditions for them to be called.
- o Ideally, the Representation Clauses field should be empty to promote the highest level of machine independence and thus reusability. However, if this is not possible, it should contain address clauses linking entries to hardware interrupts with a complete commented explanation of conditions necessary for each interrupt to occur.

II-9.2.3 Task Bodies

G9-5: Secondary unit (subunit) and first-level package body nested task bodies should have a standard format including regions for revision history, purpose, associated documentation, assumptions/resources required, side effects, diagnostics, packages, data declarations, operations and algorithmic code.

(Supports: 14, standard format; recommended information and organization supports 11, documentation for findability, 9, readability; 1, interface clarity; 6, scaffolding).

By secondary unit we mean a task body subunit that corresponds to a task body stub that appears in a secondary unit package body. These subunits are at the "library level," are candidates for reuse, and must be documented accordingly. Figure II9-2 shows an example format for a secondary unit or first-level package-nested task body. Such a body should contain at least this information, arranged in any reasonable way.⁽¹⁾ While it is important that all information in the template be available, it is not crucial that this information be stored within source code. It is important, however, that the format in which information is kept be the same for every secondary unit or first-level task body in a particular library. In this guidebook, we include all information in the template.

Figure II9-2 shows ten fields, each with one or more subfields. See the discussion for guideline G6-6 for a general description of the template fields. Additions or clarifications about the Revision History, Assumptions/Resources Required, and Algorithmic Code fields are necessary for task bodies, however.

(1) In extreme cases, this may require use of multiple fields and/or subfields with the same name. If this is necessary, we recommend using numbered indices to indicate successive fields/subfields (e.g., Subprograms (1), Subprograms (2), etc.).

```
Task body <task_simple_name> is
```

```
-- Revision History:
-- Purpose:
--   Explanation:
--   Keywords:
-- Associated Documentation:
-- Assumptions/Resources Required:
-- Side Effects:
-- Diagnostics:
-- Packages:
-- Data Declarations:
--   Types:
--   Objects:
-- Operations:
--   Subprograms:
--   Tasks:
-- Algorithm:
```

```
begin
```

```
[exception
]
```

```
end [<task_simple_name>;
```

Figure II9-2: Task Body Template

- o The Revision History field only applies to task body subunits since task body program units cannot be library units themselves. No Revision History field is necessary for task bodies contained directly within secondary unit package bodies.
- o The Assumptions/Resources Required field must contain information about the use of shared variables.
- o The Algorithm field must contain accept statements corresponding to all entries specified in the corresponding task specification. This is required by the Ada language. Commented information further clarifying entry (and entry parameter) semantics that may serve to distinguish one task body from another should be provided. The Algorithm field should also contain handlers for exceptions associated with tasking such as TASKING_ERROR if other tasks are called from the current task. These exception handlers support reusability characteristic #6 on scaffolding.

G9-6: Write task bodies to effectively handle interaction with/effects on their environment; use SHARED variables.

A Guidebook for Writing Reusable Source Code in Ada

(Supports: 3, minimal interference with the environment).

The discussion for the corresponding guideline, G6-7, assumes the absence of tasking in the executing environment for reusable software. However, consider a situation in which one or more environmental parameters are referenced and/or changed by multiple concurrently executing tasks. This significantly complicates matters. Ada does provide some assistance in the form of shared variables (assuming the particular implementation of Ada in use supports the pragma SHARED). If the pragma SHARED is not implemented, updating of environmental parameters should be strictly forbidden. Otherwise, the pragma SHARED should be used on appropriate object declarations and be fully documented, as in Example 9-a.

Example 9-a:

```

Package A is
  X : Integer := 20;
  pragma SHARED (X);

  -- X will be updated by tasks (including B,C) that modify the
  -- state of the object abstraction implemented by package A (e.g.
  -- data structure X).

  Task B is
    entry E1;
    .
    .
  end B;

  Task C is
    entry E2;
    .
    .
  end C;
end A;

Package Body A is
  Task Body B is
    begin
      accept E1 do
        X := X + 1;      -- update visible state variable
        .
      end E1;
    end B;

  Task Body C is
    begin
      accept E2 do
        X := X + 5;      -- update visible state variable
        .
      end E2;
    end C;
  end A;

```

Another facet of interaction with the environment is interaction with other tasks. This involves the rendezvous synchronization mechanism and entry call/accept statement pairs. Accept statements should contain the minimum amount of code necessary for execution during task synchronization. This is important since the calling task is suspended while this code executes.

A Guidebook for Writing Reusable Source Code in Ada

G9-7: Write task bodies with one normal exit or termination point and a grouped set of abnormal exits via exception handlers.

(Supports: 14, standardization; 9, readability; 12, modifiability).

In Ada, tasks can be activated with no "normal" exit or termination condition. However, we do not recommend this since it implies some environmental action to terminate the task and dependent program units. A normal exit in tasks can either be a TERMINATE alternative in a select statement or simply the end of a task's algorithmic code. We recommend only one normal exit. This exit should be documented. We prescribe exception handlers with at least an OTHERS alternative at the end of all task bodies to handle abnormal exists (other than task abortion from elsewhere through use of an abort statement; We will discuss abort statements later in this chapter). See Guideline G6-8. Tasks that make entry calls on other tasks should include a TASKING_ERROR alternative in their exception handlers to handle cases where the called task has completed or has become abnormal before the time of call.

G9-8: Write task body accept statements to pass results back to callers of the task rather than using results to effect task function.

(Supports: 5, minimize side effects).

This guideline corresponds directly to guideline G6-9. See G6-9 for further explanation.

II-9.2.4 Task Types and Task Objects

G9-9: Use task types to define reusable operations on data and task objects to implement particular (distinct) instances of these operations.

(Supports: 4, object-oriented software).

Consider Example 9-b.

Example 9-b:

(i)	(ii)
Package A is	Package A is
Type B is ...;	Type B is....;
Task C is	Task type C is
entry D (X:B);	entry D (X:B);
end C;	end C;
end A;	end A;

In Example 9-b (i), one task object named C is declared as an operation on data of type B. Even though package A and task C within it may be referenced by multiple units of software in an Ada program, all references will be to one task. This is different from the case when A.C is a reentrant subprogram and each client software unit gets the equivalent of a "local copy" of A.C for its own (sequential) use. In Example 9-b (ii), we illustrate the use of task types. In Example 9-b (ii), each client software unit referencing package A and task type C will in effect get one "local copy" of a task for each task object of type A.C it defines. The difference between this and the case where A.C is a subprogram is that multiple "local copies" of operations of type A.C can be defined within client software units and activated to execute in parallel rather than sequentially from "within" these units.

In general, multiple tasks operating in parallel on one data structure is quite different from one task operating on the data structure. Writers of reusable tasks need to specify the intended use for task types and objects of these types carefully. Users of tasks need to carefully follow developers' wishes, or at least understand these tasks enough to recognize the effects of modifications necessary for reuse.

II-9.2.5 Entries, Entry Calls, and Accept Statements

G9-10: Exploit entry formal parameter modes to clarify task interface semantics.

(Supports: 1, interface clarity).

Since entry formal parts are identical to subprogram formal parts, see the explanation of guideline G6-10.

G9-11: Use default entry parameters to generalize the context of a reusable task; write complete task/entry specifications.

(Supports: 10, balance between generality and specificity).

This guideline directly corresponds to guideline G6-11. See guideline G6-11 for further explanation.

G9-12: Group all default parameters in entry parameter specifications at the end of the specifications.

(Supports: 14, standardization; 3, minimal interference with the environment; 12, ease of modification).

This guideline directly corresponds to guideline G6-12. See guideline G6-12 for further explanation.

A Guidebook for Writing Reusable Source Code in Ada

G9-13: Use named parameter associations for calls to task entries with greater than three parameters or in any case for interface clarity.

(Supports: 1, interface clarity; 9, readability).

This guideline directly corresponds to guideline G6-13. See guideline G6-13 for further explanation.

G9-14: Minimize entry overloading.

(Supports: 9, readability; 13, insulation from the environment).

This guideline directly corresponds to guideline G6-14. See guideline G-14 for further explanation.

II-9.2.6 Delay Statements, Duration, & Time

Because the simple expression in delay statements must be of the predefined type DURATION from package standard, we do not recommend writing software that hides package standard. If a numeric literal is used as a delay expression, the hiding of the package standard type DURATION is no problem. However, for the sake of reusability, modification of the expression to use objects of type DURATION would be impossible if standard.DURATION were hidden.

Guideline G8-4 deals with hiding package standard or identifiers, etc. contained within it.

II-9.2.7 Select Statements

G9-15: Write all select statements with an else part or include a handler for the PROGRAM_ERROR exception at the end of the enclosing task block.

(Supports: 6, scaffolding).

The Ada LRM [DOD83] states,

"The exception PROGRAM_ERROR is raised if all [select] alternatives are closed and there is no else part [in a select statement]."

Considering the "changing environment" for reusable software, it behooves task developers to build in scaffolding to handle the unexpected case where all accept alternatives of an accept statement are closed. Including an else part in select statements, as in Example 9-c (i), will ensure that the PROGRAM_ERROR exception is not raised. Alternatively, an exception handler can be provided to handle this unexpected situation as in Example 9-c(ii).

Example 9-c:

```

-----
(i)                                ii)

Task Body A is                    Task Body A is
select                             select
  when B =>                         when B =>
    accept E1;                     accept E1;
  or                               or
  when C =>                         when C =>
    accept E2;                     accept E2;
  else                             end select;
    -- report unexpected          exception
    -- situation                 when PROGRAM_ERROR =>
  end select;                     -- report unexpected situation
end A;                            end A;
-----

```

II-9.2.8 Priorities

G9-16: Minimize use of task priorities or modify priorities accordingly when composing tasks with other tasks for the sake of reuse.

(Supports: 3, minimal interference with the environment).

Assume pragma PRIORITY is provided by the Ada implementation in use and it is used to specify the priority of a task relative to other tasks in an "original application." Now suppose this task is reused with other tasks each with their own priorities specified with the pragma PRIORITY. The relative priority of the original task is possibly no longer valid. In a system that automatically composes source code, we recommend priorities not be used (unless inspection and respecification of priorities is automated). For manual composition of source code, priorities for tasks may be used but work will be required to make sure these priorities remain meaningful in the new environment for the tasks concerned. Any use of priorities should be well documented.

II-9.2.9 Abort Statements

G9-17: Minimize use of abort statements.

(Supports: 3, minimal interference with the environment).

Use of abort statements on a task causes the named task to become abnormal, which prevents further rendezvous with it. Furthermore, any task in the environment that depends on the named task also becomes abnormal unless its

A Guidebook for Writing Reusable Source Code in Ada

execution has already completed. Exception handlers that may be contained in the bodies of aborted tasks will not be executed so that "planned" error handling/scaffolding code will be bypassed. The Ada LRM [DOD83] states

"An abort statement should be used only in extremely severe situations requiring unconditional termination."

Because of the adverse and extreme environmental impacts of abort statements and our hope that use of the reusability guidelines contained in this guidebook will facilitate writing of source code without "severe situations," we do not recommend use of abort statements.

II-9.3 GUIDELINE/CHARACTERISTIC CROSS REFERENCE

Figure II9-3 provides a cross reference between the reusability guidelines presented in this chapter and the characteristics in Chapter I-2.

Reusability Characteristic															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Guideline															
G9-1							X					X			
G9-2	X			X				X				X	X		
G9-3		X													
G9-4	X								X		X			X	
G9-5	X					X			X		X			X	
G9-6			X												
G9-7									X			X		X	
G9-8					X										
G9-9				X											
G9-10	X														
G9-11										X					
G9-12			X									X		X	
G9-13	X								X						
G9-14									X				X		
G9-15						X									
G9-16			X												
G9-17			X												

Figure II9-3: Guideline/Characteristic Cross Reference

II-10 PROGRAM STRUCTURE AND COMPILATION ISSUES

II-10.1 ADA SUMMARY

An Ada program is a set of one or more compilation units compiled together or separately in an appropriate order. Compilation units are subprogram declarations or bodies, package declarations or bodies, generic declarations or bodies, generic instantiations, or subunits which are the body of a subprogram, package, task or generic unit declared in another compilation unit. Previously compiled units can be referenced in subsequent compilations of units using a "with" context clause. "Use" context clauses can be used in conjunction with "with" clauses to achieve direct visibility of names declared within compilation units. Enforcement of language rules requires the existence of a program library which contains appropriate information about compilation units. Elaboration of all library units needed by a main program is done before execution in an order consistent with library unit dependencies and other rules, but can be effected by use of a pragma.

II-10.2 GUIDELINES

The guidelines for constructing reusable programs or compilation units mainly support the findability and fit-to-be-reused metacharacteristics. Fit-to-be-reused considerations center on being able to easily group software modules together and modify them if necessary. While the guidelines in this chapter have some bearing on the understandability of reuse candidates, guidelines addressing understandability explicitly can be found in Chapters of Section II that deal with specific compilation units or language constructs.

The guidelines in this Chapter state how reusable components or parts should be structured in Ada. G10-1 expresses a standard form for "catalogued" reusable components. G10-2 and G10-3 refine G10-1 specifying exactly what portions of component internals are reusable. G10-4 through G10-7 explain how to write reusable components for maximum readability and modifiability, and minimal negative environmental impact. The most important point made in this chapter is that we have chosen packages as our "unit of reusability."

II-10.2.1 Compilation Units - Library Units

There are, in general, two kinds of reusable software parts - directly reusable parts and indirectly reusable parts. Directly reusable parts are those whose behavior or effect is catalogued, that is, "advertised" in the catalog⁽¹⁾ of reusable software that developers use to determine what software parts are available for reuse. Directly reusable parts are what developers

(1) A catalog can be an automated software repository's classification scheme, a list of component names and descriptions on paper, or even a rumor the developer hears from a colleague down the hall.

search for and choose. Indirectly reusable parts support directly reusable parts; they provide the environment, the ancillary definitions and data that the directly reusable parts need in order to perform correctly. In the ideal case, indirectly reusable parts are incorporated into the program under construction automatically by a software base management system.

II-10.2.1.1 Directly Reusable Parts

Reusable components should be objects. As abstractions, objects have properties (data) and allowable operations on this data. The Ada package should be the realization or concrete implementation of the object abstraction. It should contain basic declarations as defined in Ada (e.g., type and data object declarations and subprogram/task specifications). Types and data objects/variables implement data; subprograms/tasks implement operations. Packages bundle these things up nicely.

G10-1: Use library unit package specifications as the encapsulation mechanism for directly reusable software (i.e., data and operations on the data).

(Supports: 4, object oriented software; 14, standardization).

As described in Chapter II-7, Ada package specifications have public and private declarative parts. Chapter II-7 also recommends a template for package specifications. The basic declarations of an abstract object's visible structural properties should be grouped together at the beginning of the public declarative part in a clearly-marked "Data Declarations" region (see Figure II10-1). This region is one of several fields in the Chapter II-7 package specification template.


```

With <unit_simple_name>;          --reference to library unit
Package <package_simple_name> is

  --Data Declarations: includes types/objects,...

  --Operations:  includes procedure, function, task specifications

  private
  .
  .
end;
```

Figure II10-1: Package Specifications Implement Directly Reusable Software Parts

Basic declarations can include (1) type, (2) object, (3) exception, (4) number, (5) subtype, (6) deferred constant, and (7) renaming declarations, or (8) package specifications containing the above declarations and other basic declarations not previously mentioned. Following the Data Declarations region, an "Operations" region containing procedure, function, and task specifications should be written. This corresponds to another field in the Chapter II-7 template for package specifications. It is the package specification that contains the interface to reusable objects. Context for operations declared in a package specification is important. Context is not only data in the Data Declarations region but also data and operations referenced in Ada context clause library units and package Standard. Package specifications containing specifications of operations on data are directly reusable software parts. See Figure II10-2.

```

With A;           --library unit A is indirectly reused
Package B is
.
.               --public part of package specification for B is directly
.               --reusable
private
.
.               --private part is indirectly reusable
end B;

Package body B is
.
. Procedure C (X:D) is separate;
.
.               --package body B is indirectly reusable
end B;

Separate (B)
Procedure C (X:D) is
.
.               --subunit for C is indirectly reusable
end C;

```

Figure II10-2: An Example Of Directly and Indirectly Reusable Software Parts

G10-1 states that directly reusable software parts are package specifications containing specifications of operations on data. Examples 10-a and 10-b present a case where data and operations on data are not so easily packaged together.

Example 10-a:

<pre> With E; Package A is Type C is... Procedure B (X:C; Y:E.F); end A; </pre>	<pre> With A; Package E is Type F is... Procedure G(X:F; Y:A.C); end E; </pre>
-------------------------------------------------------------------------------------	------------------------------------------------------------------------------------

The packages in Example 10-a, taken together, are illegal in Ada because they are mutually dependent. At first glance, a solution would be to define a new package that combines all the operations and data. This is recommended. However, if this is not desirable for other reasons such as modularity or understandability, a package consisting of the data from packages A and E could be declared and referenced from both packages A and E, as in Example 10-b.

Example 10-b:

```

Package Common_Data is
  Type C is...
  Type F is...
end Common_Data;

with Common_Data;
Package A is
  --Types C,F are defined in package Common_Data.

  procedure B (X: Common_Data.C, Y: Common_Data.F)...;
end A;

With Common_Data;
Package E is
  --Types F,C are defined in package Common_Data.

  procedure G (X: Common_Data.F, Y: Common_Data.C)...;
end E;

```

The important point here is that library unit packages are our "unit of reusability" with package specifications as the standard unit for directly reusable software parts. It is the specifications of operations on data as well as data contained in these packages that are directly reusable. In Example 10-b, packages A and E are directly reusable. Procedures B and G within packages A and E operating on their appropriate data are directly reusable. Package Common_Data supports these directly reusable components and the packages that contain them but is not itself directly reusable. It is indirectly reusable. Packages A and E encapsulate implementations of operations associated with an abstract object or part of the object. The temptation is to write procedure specifications B and G as separately compilable library units and not within a package specification. However, we recommend writing them as in Example 10-b. Another case where use of separate or library unit subprograms might be postulated is for operations on predefined data (e.g., integers). However, again we recommend these operations be packaged up with comments describing the data used.

A Guidebook for Writing Reusable Source Code in Ada

G10-2: Only "first level" nested non-package entities in library unit package specifications form the basis for "catalogued" directly reusable objects/software.

(Supports: 2, appropriate level of interface).

Ada packages can be nested to any level allowed by a compiler implementation, and nesting can be used as desired for implementing reusable components. However, for ease of "cataloging" there should be a practical limit to the level of nesting of packages that encapsulate reusable software. G10-2 simply states that only first-level data and specifications for operations on data form the basis for reusable software and are "catalogued." Data and operations within nested packages are not catalogued as reusable even though they are accessible to client programs according to the Ada language definition. Nesting can easily complicate the environment or context for reusable software. For example, nesting provides an environment for declaration order, information hiding, and visibility rules which is hard to reuse and to understand, and in which operations and data are hard to classify. Classifying only entities that are visible at the first level as reusable operations on data in context will avoid this complication.

Example 10-c:

```
Package A is
  Package C is
    Type D is...;
    Procedure Sort_D (X:D; Y: out D)...;
  End C;
  Type B is...
  Procedure Sort_B (X:B; Y:C.D; Z:out B)...;
End A;
```

To illustrate this, in Example 10-c Procedure Sort_B is catalogued as a reusable operation on data of type B (which is also catalogued) in the context of package A; Package C is not declared as a service to the outside world as Sort_B is. Thus, neither package C nor its contents are catalogued. Nested packages and their contents should not be visible to the outside world for the purposes of reusability even though they are in Ada.

Now, why would Package C be declared within Package A? One reason might be to group logical entities within A that are a service to A. On the other hand, if this package is to constitute an encapsulation of reusable software, useful to more than package A, it should be broken out as a separate library unit and be referenced in a context clause in package A. Note that although type D and procedure Sort_D are not catalogued, procedure Sort_B uses type D from package C. This is acceptable. The operation Sort_B is directly reusable while the data it operates on may be indirectly reusable.

II-10.2.1.2 Indirectly Reusable Parts

G10-3: Use secondary unit package bodies, package specifications containing only data, and subunits corresponding to first-level package body nested stubs as the encapsulation mechanism for indirectly reusable software.

(Supports: 4, object-oriented software; 14, standardization).

This guideline, along with G10-1, states that all reusable Ada software should be written in terms of packages. In particular, subprograms (with the exception of main subprograms) and tasks should be written either directly within the declarative parts of library unit packages or in that context through the use of body stubs. In Ada, main programs must not be contained in packages. However, we do not treat them as reusable. It is the library unit packages they reference that are reusable. Secondary unit (library unit) package bodies are indirectly reusable. Subprograms and tasks in the context of library unit packages are indirectly reusable. Library unit package specifications containing only data are indirectly reusable as well. See Figure II10-2 to clarify the distinction between directly and indirectly reusable software parts.

II-10.2.2 Context Clauses

G10-4: "With" clauses on package specifications should reference only data needed in specifications. "With" clauses can be used freely on package bodies as needed.

(Supports: 13, insulation from the environment).

Clearly, an indirect reference to data needed in a package body by a with clause on the body's corresponding specification is assuming this environment for all possible bodies. For reuse, a natural occurrence is to replace bodies and to require different environments for them. Different bodies for the same specification may need data from different library units. An attempt to change bodies using "with" clauses on the specification only may require a context clause change on the specification or extra "with"ed library units in an attempt to be general. See Example 10-d.

Example 10-d:

```
With A;  --information from A needed only in B's body
Package B is
.
.
.
end B;

Package Body B is
.
.
.
end B;
```

Assume in Example 10-d that information from package A is needed in package B's body and not in its specification. If, in order to reuse package B, package B's body is replaced by a new body not requiring information from A, then the reference to A in package B's specification should be removed. Leaving the reference to A on package B's specification would be inefficient, unnecessary, and confusing to readers, and thus is not recommended. This would all be unnecessary if the reference to A was originally written in a context clause on package body B.

II-10.2.3 Subunits of Compilation Units

G10-5: Use subunits to achieve modularity and ease of recompilation.

(Supports: 9, readability; 12, modifiability).

The ability to use subunits to implement the body of a program unit that is declared within another compilation unit permits top down hierarchical development and facilitates program modularity, readability and modifiability. Modularity and readability are improved by allowing a body stub to take the place of the proper body of a program unit, as in Example 10-e.

Example 10-e:

```

Package A is
  procedure B;
end A;

Package Body A is
  .
  .
  .
  procedure B is separate;
  .
  .
end A;

separate (A)      --subunit
procedure B is
  .
  .
  .
end B;

```

If the body of procedure B is long, and if package body A contains a number of relatively long procedure bodies, the body of package A will become very large and hard to understand and change. When the body of procedure B is written as (1) a body stub within package body A and (2) a separate subunit, package body A will be of reasonable length. Use of a body stub and subunit for procedure B increases modifiability in that a change in B (except in its parameters) will necessitate recompilation of its subunit only. Package A's entire body would need to be recompiled if procedure body B were written directly within package body A and B is changed. Subunit B could easily be replaced with another subunit if reuse of the specification for B and thus package A required it.

Subunits can be reused in only one place, the place of the corresponding body stub or placeholder previously compiled. A subunit has an environment of visible identifiers, etc. available to it as if it was declared in place of its corresponding stub. Therefore, as stated above, a subunit is an indirectly reusable software part. This is not to belittle the importance of subunits, however. If used as described above, subunits are extremely important to reusability. We see the need for both directly and indirectly reusable software parts. Part of developing reusable software is determining when something should be structured as an indirectly reusable part, as the separate subunit is in this example. An important facet of reusable software development is following the appropriate guidelines when creating indirectly reusable parts.

II-10.2.4 Order of Compilation

G10-6: Use separate compilation and separate specifications and bodies to achieve modularity and ease of recompilation.

(Supports: 7, separation of specification from body; 12, modifiability).

Because of Ada's visibility rules and dependencies among library units referenced in context clauses, Ada prescribes an order of compilation for a multi-part program unit. For example, a change to a package specification necessitates its recompilation as well as the recompilation of all library units that reference it in a context clause. Any way source code can be written to minimize recompilations necessitated by changes to it will enhance reusability. This includes not only use of subunits as mentioned above but also use of separate compilation of specifications and bodies. Separate bodies can be changed and/or replaced and recompiled without affecting their corresponding specifications.

II-10.2.5 The Program Library

We feel that creative use of Ada program libraries can provide significant support for reusability. For example, use of multiple domains within a library or multiple libraries can facilitate software reuse where multiple application areas are concerned. The library system can serve as an aid to cataloging and retrieving software parts for reuse. Examine the capabilities of your Ada implementation to determine the extent to which it provides such support.

II-10.2.6 Elaboration of Library Units

G10-7: Minimize use of Pragma Elaborate.

(Supports: 3, interference with the environment (minimization of side effects); 13, insulation from the environment).

Library unit specifications and bodies needed by a main program are elaborated along with elaboration of library units referenced by these library units and so on before execution of the main program. Elaboration order depends on the relationships or partial orderings established by "with" clauses and subunits. The pragma Elaborate may be needed if this ordering is not sufficient to ensure elaboration of each library unit body before elaboration of any other compilation unit whose elaboration depends on prior elaboration of one of these bodies. However, we recommend limiting the use of this pragma since it complicates the elaboration order issue and may lead to there being no consistent elaboration order when groups of modules are combined for reuse.

II-10.3 GUIDELINE/CHARACTERISTIC CROSS REFERENCE

Figure II10-3 provides a cross reference between the reusability guidelines presented in this chapter and the characteristics in Chapter I-2.

Program Structure and Compilation Issues

	Reusability Characteristic														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Guideline															
G10-1				X											X
G10-2		X													
G10-3				X											X
G10-4													X		
G10-5									X			X			
G10-6							X					X			
G10-7			X											X	

Figure II10-3: Guideline/Characteristic Cross Reference

II-11 EXCEPTIONS

II-11.1 ADA SUMMARY

II-11.2 GUIDELINES

II-11.2.1 EXCEPTION HANDLERS

II-11.2.2 RAISE STATEMENTS

II-11.2.3 EXCEPTION HANDLING

--during execution of statements, elaboration of declarations, task execution

II-11.2.4 Suppressing Runtime Checks

II-11.3 Guideline/Characteristic Cross Reference

II-12 GENERIC UNITS

II-12.1 ADA SUMMARY

Generic units are program units of two types: either generic subprograms or generic packages. These units serve as templates for nongeneric subprogram or package program units. These templates commonly involve parameterization but parameterization is optional. Non-generic instances of an Ada generic unit are obtained by a process called instantiation. Generic declarations declare generic units and consist of subprograms and/or package specifications which have generic formal parts where generic formal parameters (i.e. types, objects and subprograms) can be specified. An instance of a generic unit, a declaration in its own right obtained through instantiation of the unit, associates generic actual parameters with formal parameters contained in the generic unit declaration. A set of matching rules applies to this association of parameters. Generic bodies are subprogram or package bodies that correspond respectively to generic subprogram or package declarations. The syntax of these bodies is no different than for nongeneric bodies. However, as for generic declarations, the bodies act as templates for bodies obtained through generic instantiation (of a corresponding specification).

II-12.2 GUIDELINES

This chapter's guidelines for writing Ada generic units support all of the reuse metacharacteristics described in Section I and most of the characteristics they imply.

II-12.2.1 Generic Declarations

G12-1: Use generic program units (i.e., packages and subprograms) to effectively parameterize reusable software parts.

(Supports: 10, generality through parameterization; 12, use with minimal modification; 9, readability).

One main form of parameterization in Ada is use of subprogram parameters. However, subprogram parameters are limited to objects and their values. Generic formal parameters, in addition to objects/values, can be types and/or subprograms as well.

[BOOCH83b] states:

"By no means ignore the use of generics. If you thought that packages were a powerful structure, just wait till you grasp the power of generics. Generics make possible the production of truly reusable software components, and their proper design can mean that

the cost of the production of a given unit can be amortized over its many applications, in addition to facilitating the reliability and understandability of a system.

"There are two basic models I use when designing generic units, one rather 'classical' and the other not so classical. In the classic case, generic units may be designed to form templates whose instantiations apply to a large class of objects. Traditional data structures such as stacks, queues, lists, etc. are good candidates for generic units, as are searching, sorting and some mathematical operations. For example:

```
generic
  type ELEMENT is private;
package FIFO_QUEUE is
  type KIND is private;
  procedure INSERT (E : in ELEMENT; ON : in out KIND);
  procedure REMOVE (E : out ELEMENT; FROM : in out KIND);
  ...
end FIFO_QUEUE;

generic
  type INDEX is (<>);
  type COMPONENT is private;
  type ARY is array (INDEX) of COMPONENT;
  with function "<" (LEFT, RIGHT : in COMPONENT) return BOOLEAN;
procedure QUICK_SORT (A : in out ARY);

generic
  type ANGLE is digits <>;
  type RESULT is digits <>;
package TRANSCENDENTAL_FUNCTIONS is
  function COS (A : in ANGLE) return RESULT;
  function TAN (A : in ANGLE) return RESULT;
  ...
end TRANSCENDENTAL_FUNCTIONS;
```

"Now, I'll admit that writing a correct body for a generic is not a trivial exercise, but the judicious use of attributes makes the task possible. When architecting the generic part of such a unit, I ask myself the following question: how can I generalize this component to operate across a class of types? The answer can give me a guideline on extracting the generic nature of a data structure or an algorithm."

I will discuss Booch's second generics model in the context of guideline G12-2 below.

Generics are invaluable because of the parameterization capabilities they provide the Ada developer and reuser. Treating the types of objects and operations on them as parameters opens up another dimension to the possible applications of a particular piece of source code. Generics provide a

well-defined way to modify source code in order to reuse it. No recompilations are necessary if all tailoring to source code can be done by supplying generic actual parameters in a generic instantiation.

G12-2: Use generic program units to precisely specify module interfaces/ imports and exports.

(Supports: 1, interface clarity; 2, appropriate abstract level; 8, low coupling; 13, insulation from the environment).

[BOOCH83b] describes his second model for using generics in this way...

"The second model for using generics is less classical - let me lead up to its description. If, in a library unit, I 'with' another unit, then I have essentially imported all the visible facilities of that withed unit. The important word to recognize here is all. Package specifications have essentially an open scope, and once I 'with' a unit, there is no convenient way to limit the use of a visible part. Furthermore, in order to understand the functionality of a given unit, it is necessary to examine the visible part of all withed units. Neither of these situations is necessarily very desirable.

"In such cases, we may apply generic units to more precisely specify module interfaces, and hence provide control over both imports and exports relative to a module. In formulating such a unit, the essential question to ask is NOT what general classes of types may be applied to this module, but rather, what facilities does this module need in order to fulfill its purpose. As an example, consider building a system for plotting simple graphs. At the lowest level, we may abstract a point on a graph as follows:

```
generic
  type ELEMENT is digits <>;
package POINT is
  type KIND is private;
  procedure SET (K : out KIND;
                X : in ELEMENT;
                Y : in ELEMENT);
  function X (K : in KIND) return ELEMENT;
  function Y (K : in KIND) return ELEMENT;
private
  ...
end POINT;
```

"This is an example of a generic of the first kind. In this case, all a given point needs to 'know' is its accuracy. We may build upon this abstraction by defining the specification of the graph generator:

```
generic
  type X_VALUES is digits <>;
  type Y_VALUES is digits <>;
  type COORDINATE is private;
  with function X(C : in COORDINATE) return X_VALUES;
  with function Y(C : in COORDINATE) return Y_VALUES;
package GRAPH is
  procedure SET_UP;
  procedure SET_TITLE (TO : in STRING);
  procedure ADD (C : in COORDINATE);
  procedure GENERATE;
end GRAPH;
```

"This is an example of a generic of the second kind. Instances of the package GRAPH need only have knowledge of the range of values over the domain and range, and some characteristics of points. Notice in this example, that in order to understand the functionality of GRAPH, we need not know the details of any external entity (although a little documentation certainly wouldn't hurt). Still, the unit is generalized, since it may be instantiated across multiple kinds of points. We have thus effectively defined an import list for the unit. Furthermore, we may now use these units in layers of abstractions, wherein the layers are relatively independent. For example:

```
type VALUE is digits 10;
package VALUE_POINT is new POINT (ELEMENT => VALUE);
package VALUE_GRAPH is new GRAPH (X_VALUES => VALUE,
                                   Y_VALUES => VALUE,
                                   COORDINATE => VALUE_POINT.KIND,
                                   X          => VALUE_POINT.X,
                                   Y          => VALUE_POINT.Y);
```

"One caveat must be mentioned: depending upon the subject compiler, this use of generics to form layers of abstractions may result in a space penalty. Of course, this penalty must be weighed against increased control of interface specification."

The key to Booch's second model for using generics is in the clarity and control that layered generic abstractions provide. Referring to his example, the point abstraction is separated from the graph abstraction and the "kind of" facilities each needs from the environment (i.e., generic parameters and/or types and subprograms). Facilities provided to the environment are clearly specified. The emphasis here, however, is on facilities required from the environment. This separation of object abstractions provides a level of simplicity to the user and the fact that they are layered corresponds directly to users' intuition.

It is interesting that although conceptually package graph depends on package point, this dependency does not manifest itself directly in a context (i.e., a with) clause relationship. Thus, packages point and graph can be relatively independent. The dependency between the two packages is a dependency between

instantiations as Booch illustrates with packages `value_point` and `value_graph`. In the instantiations of packages `point` and `graph`, generic actual parameters provide an explicit indication as to information needed from the environment and just how the two packages are related (i.e., package `value_graph` needs type `value_point.kind` and functions `value_point.x` and `value_point.y`). The fact that package `graph` did not need to "with" package `point` and only the needed subset of capabilities of package `point` were used by the instantiation of package `graph` (i.e., procedure `value_point.set` was unused), Booch was able to control or limit use of the visible part of package `point`.

Thus, interfaces are clear, package `graph` is insulated from the environment (e.g., package `point`), and coupling between packages `graph` and `point` is minimized. Also, as for subprogram parameters, all generic parameters should be written at an appropriate abstract level for the function of the generic.

G12-3: Use generics to allow specification of multiple instances of reusable software as compared to reuse of one shared instance.

(Supports: 15, right abstraction for the application).

In the non-generic world, one can declare a package with an instance of an encapsulated object (i.e., a variable) to be reused. By encapsulated, we mean this variable is declared in a package body with subprogram (or task) interfaces specified in the corresponding package specification. (See Chapter II-7). We have good reasons for doing this, one of which is to provide standard interfaces to reusable object instances. Every reference to such an object is indirect. The package being reused is one package that contains one set of subprograms (and tasks) operating on an instance of one object. If only one such instance is needed, use of generics is not required. If more are needed, use of generics is required. If we want these multiple object instances to be different, different generic parameters are required for each instance. However, even in the case where multiple, identical reusable object instances are required, generics provide programmers with an effective tool as well. See Example 12-a.

Example 12-a:

```

generic
  type Element is private;
  package Stack is
    procedure Add (X : in Element);
    procedure Remove (X : out Element);
  private
    ...
  end Stack;

  package body Stack is
    type Stack_Type is...;
    My_Stack : Stack_Type;
    procedure Add ...;
    procedure Remove ...;
  begin
    ...
  end Stack;

  package Games_People_Play is
    procedure Duelling_Stacks;
    ...
  end Games_People_Play;

  with Stack;
  package body Games_People_Play is
    procedure Duelling_Stacks is
      package Stack1 is new Stack (INTEGER);
      package Stack2 is new Stack (FLOAT);
    begin
      ...
    end Duelling_Stacks;

  end Games_People_Play;

```

-- two instances of stack
 -- package are required
 -- since two encapsulated
 -- stack object instances
 -- are required.

In Example 12-a, first a generic stack package is declared. Next, the package Games_People_Play is written containing procedure Duelling_Stacks. Duelling_Stacks needs two instances of stacks. A variable declaration for a stack is encapsulated in the body of package Stack with an indirect interface in its corresponding specification. By making package Stack generic, multiple, different instances of stacks can be provided by generic instantiations, as in the body of package Games_People_Play which is in the body of procedure Duelling_Stacks. In the case where two identical encapsulated stacks were required, two instantiations of package Stack with identical "element" param-

ters could have been used. Without generics, two (nonparameterizable) instances of the same stack package would need to be declared (one with a different name) to achieve this effect.

G12-4: Use basetypes rather than subtypes to specify the type of a generic formal object or generic formal subprogram parameter or result types.

(Supports: 9, readability).

This guideline comes directly out of the Ada Reference Manual [DOD83]. [DOD83] states:

"The constraints that apply to a generic formal object of mode in out are those of the corresponding generic actual parameter (not those implied by the type mark that appears in the generic parameter declaration). Whenever possible (to avoid confusion) it is recommended that the name of a base type be used for the declaration of such a formal object. If, however, the base type is anonymous, it is recommended that the subtype name defined by the type declaration for the base type be used."

[DOD83] then goes on to say:

"The constraints that apply to a parameter of a formal subprogram are those of the corresponding parameter in the specification of the matching actual subprogram (not those implied by the corresponding type mark in the specification of the formal subprogram). A similar remark applies to the result of a function. Whenever possible (to avoid confusion), it is recommended that the name of a base type be used rather than the name of a subtype in any declaration of a formal subprogram. If, however, the base type is anonymous, it is recommended that the subtype name defined by the type declaration be used."

G12-5: Library unit and first-level package nested generic unit declarations should have a standard format, including a region for a description of generic parameters as well as standard information required for non-generic subprogram and package declarations.

(Supports: 14, standard format; recommended information and organization supports 11, documentation for findability, 9, readability).

Figures II12-1 and II12-2 show example templates for library unit and first-level generic declarations. These declarations should contain at least this information, arranged in any reasonable way.(1) While it is important

(1) In extreme cases, this may require use of multiple fields and/or subfields with the same name. If this is necessary, we recommend using numbered indices to indicate successive fields/subfields (e.g., Subprograms (1), Subprograms (2), etc.).

that all information in the templates be available, it is not crucial that this information be stored within source code. It is important, however, that the format in which information is kept be the same for every library or first-level generic unit in a particular library. In this guidebook, we include all information in the template.

```
generic
.
.
.
procedure or function <subprogram_name>
    <parameter_list> [return] <typemark>;

-- Revision History:
-- Purpose:
--   Explanation:
--   Keywords:
-- Generic Parameter Description:
-- Parameter Description:
-- Associated Documentation:
```

Figure II12-1: Generic Subprogram Declaration Template

```

generic
.
.
.
Package <package_name> is

-- Revision History:
-- Purpose:
--   Explanation:
--   Keywords:
-- Generic Parameter Description:
-- Associated Documentation:
-- Diagnostics:
-- Packages:
-- Data Declarations:
--   Types:
--   Objects:
-- Operations:
--   Subprograms:
--   Tasks:
-- Private:

End <package_name>;

```

Figure II12-2: Generic Package Declaration Template

The Generic Parameter Description field contains a brief explanation of each generic formal parameter and clarifications of parameter semantics that apply to all possible generic bodies corresponding to the declaration. Semantics peculiar to the bodies should be described in documentation for the bodies themselves. This clarification of semantics is more difficult for generic parameters than for, say, non-generic subprogram parameters in view of the additional kinds of parameters (i.e., types and subprograms) allowed.

Bodies corresponding to generic units are identical to bodies of non-generic units. See Chapters II-6 and II-7 for details. See guidelines G6-5 and G7-4 for descriptions of other template fields.

G12-6: Separate generic declarations from bodies for ease of recompilation and modification.

(Supports: 12, use with minimal modification; 7, separation of specification from body).

See the discussion for guideline G6-1 in Chapter II-6 and guideline G7-1 in Chapter II-7 for further information.

II-12.2.1.1 Generic Formal Objects

G12-7: Exploit generic formal object parameter modes to clarify interface semantics.

(Supports: 1, interface clarity).

Generic formal object parameters can have modes of in or in out. As for subprogram parameters, (see guideline G6-10), specification of parameter modes that constrain use of parameters to their intended function make interface semantics clear and avoid confusion/unexpected results.

II-12.2.1.2 Generic Formal Types

G12-8: Use generic type definitions to clarify interface semantics and module operation.

(Supports: 1, interface clarity).

A difference between generic formal parameters and subprogram formal parameters is that with generics, types as parameters must be considered. Generic type definitions specify types from one of six general classes: discrete, integer, floating point, fixed point, array and access types. Along with these types come implicit operations on objects of these types. Use generic type definitions appropriate to the functionality of corresponding generic packages and procedures.

G12-9: Use "additional" generic parameters as necessary to effect inheritance of desired operators on generic formal types.

(Supports: 4, object-oriented software).

The Ada Language Reference Manual [DOD83] states:

"For an instantiation of the generic unit, each of these operations [i.e., operations associated with type classes in generic type definitions] is the corresponding basic operation or predefined operator of the matching actual type. For an operator, this rule applies even if the operator has been redefined for the actual type or for some parent type of the actual type."

Thus, in defining a generic package or subprogram with a generic formal type as parameter, one must realize that none of the redefined operators for the type that may have been provided with the type will be inherited for use in an instantiated generic. This would tend to break down our object-oriented approach to reusable software parts since types and operations on data of these types would (somewhat unnoticeably) become separated. The solution to this problem is to define generic subprograms as generic formal parameters in

addition to generic formal type parameter(s) and require any redefined subprogram operators of concern to be explicitly "passed" to generic instantiations. See Example 12-b.

Example 12-b:

```

Package A is
  type X is .. range 1...10;
  function "+" (Y, Z : in X) return X;  -- redefines predefined
  .                                     -- integer adding
  .                                     -- operator;
end A;

generic
  type B is range <>;
  -----
  | with function "+" (E, F: in B) return B;| -- must be provided to
  ----- -- inherit any redefined adding
  ----- -- operations on objects
  ----- -- of type B

procedure C is
  D : B;
  .
  .
  .
begin
  D := D + 1;  -- "+" expected to be appropriate adding operator
               -- for objects of generic formal type B;
end C;

with A;
Package Body G is
  ...
  Procedure H is
    procedure MY_C is new C(A.X, A."+"); --appropriate adding operator on
    --objects of type X provided to
    --instantiation of C;

    begin --H
      ...
    end H;
  end G;

```

II-12.2.1.3 Generic Formal Subprograms

G12-10: Minimize generic formal subprogram parameter overloading and overloading of subprograms in generic packages.

(Supports: 9, readability; 13, insulation from the environment).

This guideline corresponds to guideline G6-14 in Chapter II-6, dealing with overloading of subprograms. In addition to readability and understandability aspects associated with subprogram overloading discussed in Chapter II-6, the discussion of binding of overloaded subprograms to their definitions is particularly relevant here. This binding depends on the context in which overloaded subprogram names are used. Overload resolution depends not only on subprogram names but on their parameters and result types (for functions) as well. In addition to context changes that may affect subprogram name binding that are not caught by an Ada compiler or user that are applicable when generics are not used, we refer to [DOD83] for another warning:

"If two overloaded subprograms declared in a generic package specification differ only by the (formal) type of their parameters and results, then there exist legal instantiations for which all calls of these subprograms from outside the instance are ambiguous. For example:

```
generic
  type A is (<>);
  type B is private;
package G is
  function NEXT(X : A) return A;
  function NEXT(X : B) return B;
end;

package P is new G(A => BOOLEAN, B => BOOLEAN);
-- calls of P.NEXT are ambiguous"
```

In the [DOD83] example above, the context of the two overloaded subprograms named NEXT is generic package G with parameters for types A and B, and any other software visible (now or in its extended environment for reuse) to G. The fact that the types A and B are generic parameters to package G and that both NEXT functions depend on these parameters makes for a potentially ambiguous situation. This is illustrated in the example. This unnecessarily complicates reuse of G and could have been avoided by declaring NEXT functions with different names. Package G puts an unnecessary constraint on its environment by requiring the functions next to operate on different parameter types or return results of differing types in order for no ambiguity to exist. What if a user wants types A and B to be the same? This would require a change to one of the package G NEXT functions.

Incidentally, generic formal subprograms as parameters can be overloaded but for the same reasons as stated above, this should be minimized.

G12-11: Minimize use of the box (is <>) notation to specify default generic formal subprograms as parameters.

(Supports: 13, insulation from the environment).

[DOD83] specifies the following use of the "box" notation:

"If a generic unit has a default subprogram specified by a box, the corresponding actual parameter can be omitted if a subprogram, enumeration literal, or entry matching the formal subprogram, and with the same designator as the formal subprogram, is directly visible at the place of the generic instantiation; this subprogram, enumeration literal, or entry is then used by default (there must be exactly one subprogram, enumeration literal or entry satisfying the previous conditions)."

This is potentially dangerous for reusability because it assumes in the default case direct visibility of a subprogram, enumeration literal, or entry. Unless this default is declared in the same program unit as the generic unit, (and in this case we recommend using named defaults), do not depend on the direct visibility of anything when it comes to the volatile environment of a reusable software module. The context of a reusable software module is expected to change from use to use. Use of the box notation puts an explicit and unnecessary requirement on this context.

II-12.2.2 Generic Bodies

G12-12: Use basic operations/attributes associated with generic formal types to provide required generality to generic bodies.

(Supports: 10, balance between generality and specificity).

Predefined language basic operations/attributes provide an effective mechanism to generalize generic bodies. This is because in many instances they apply to classes of types, not just particular types. A good example of this is the use of discrete types. (See Example 12-c.) If a generic formal type is specified as a parameter and the generic type definition involved calls for a discrete type, the body of the corresponding generic subprogram or package has at its disposal basic operations of assignment, membership tests, qualification, and a series of attributes (base, first, last, width, pos, val, succ, pred, image, value, size and address). These operations/attributes can be used independently of the particular discrete type provided as a parameter in instantiations of a generic subprogram or package. A similar case applies for other classes of generic formal types.

Example 12-c:

```
generic
  type A is (<>);
  procedure X;
  .
  .
end X;

procedure X is    --body of generic procedure X
  .
  .
begin
  For I in A'FIRST..A'LAST loop
    .                -- attributes FIRST and LAST are used
    .                -- to generalize generic body
  end loop;
end X;
```

II-12.2.3 Generic Instantiations

G12-13: Used named parameter association in actual parameter parts of generic instantiations.

(Supports: 1, interface clarity; 9, readability).

This guideline directly corresponds to guideline G6-13 in Chapter II-6 on subprograms. The difference between that guideline and this one is that we recommend named notation always be used with generic parameters and in Chapter II-6, we suggested that it be used with three or more subprogram parameters. In general, it is more difficult to look at a generic instantiation and manually associate actual and formal parameters than for subprogram parameters because generic parameters can be types and subprograms in addition to objects and values. [DOD83] states that named associations are not allowed for use in instantiations with overloaded subprogram designators. However, we have already advised against overloading in guideline G12-10.

G12-14: Use default parameters for generic actual parameters whenever possible.

(Supports: 10, balance between generality and specificity; 9, readability).

This guideline corresponds to guideline G6-11 in Chapter II-6 on subprogram default parameters. Generics allow default values for objects and subprogram parameters. The defaults for objects are much the same as for subprogram parameters. (See Chapter II-6). However, subprograms as generic parameters themselves can simplify reuse and give the user an example of a viable subprogram to use as a parameter. This makes reusable generic source code more readable/understandable.

G12-15: Create particular instantiations of generic units corresponding to common uses of reusable software.

(Supports: 15, domain of applicability).

Instantiations of generic units that are "right" for a particular application domain will be more easily used than the original generic units (uninstantiated). We feel that in many cases, as libraries of reusable code are built up for particular application areas, sets of instantiated generics will likely emerge as reusable parts in addition to the original generic units.

II-12.3 GUIDELINE/CHARACTERISTIC CROSS REFERENCE

Figure II12-3 provides a cross reference between the reusability guidelines presented in this chapter and the characteristics in Chapter I-2.

	Reusability Characteristic														
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Guideline															
G12-1									X	X		X			
G12-2		X	X					X					X		
G12-3															X
G12-4									X						
G12-5									X		X			X	
G12-6							X					X			
G12-7		X													
G12-8		X													
G12-9				X											
G12-10									X				X		
G12-11													X		
G12-12										X					
G12-13		X							X						
G12-14									X	X					
G12-15															X

Figure II12-3: Guideline/Characteristic Cross Reference

A Guidebook for Writing Reusable Source Code in Ada

I-13 REPRESENTATION CLAUSES AND IMPLEMENTATION DEPENDENT FEATURES

I-13.1 ADA SUMMARY

I-13.2 GUIDELINES

I-13.2.1 REPRESENTATION CLAUSES

I-13.2.1.1 LENGTH, ENUMERATION, RECORD, AND ADDRESS CLAUSES

I-13.2.1.2 CHANGE OF REPRESENTATION

I-13.2.2 IMPLEMENTATION DEPENDENT FEATURES

I-13.2.2.1 THE PACKAGE SYSTEM

I-13.2.2.2 MACHINE CODE INSERTIONS

I-13.2.2.3 INTERFACE TO OTHER LANGUAGES

I-13.2.2.4 UNCHECKED PROGRAMMING

I-13.3 GUIDELINE/CHARACTERISTIC CROSS REFERENCE

II-14 INPUT-OUTPUT

II-14.1 ADA SUMMARY

II-14.2 GUIDELINES

II-14.2.1 EXTERNAL FILES AND FILE OBJECTS

II-14.2.2 SEQUENTIAL AND DIRECT FILES

II-14.2.3 TEXT INPUT-OUTPUT

II-14.2.4 EXCEPTIONS IN INPUT-OUTPUT

II-14.2.5 LOW-LEVEL INPUT-OUTPUT

II-14.3 GUIDELINE/CHARACTERISTIC CROSS REFERENCE

A APPENDIX: LIST OF GUIDELINES

This appendix contains a complete list of guidelines specified in this guidebook.

- G6-1: Separate subprogram declarations and bodies for ease of recompilation and modification.
- G6-2: All reusable subprograms except a main program must be written within a library unit package.
- G6-3: Use subprogram declarations to specify interfaces to reusable objects. Use subprogram bodies to implement these interfaces and properties of the objects.
- G6-4: Write subprogram interfaces at an appropriate abstract level.
- G6-5: First-level package-nested subprogram declarations should have a standard format including regions for purpose, parameter descriptions and associated documentation.
- G6-6: Secondary unit (subunit) and first-level package body nested subprogram bodies should have a standard format, including regions for revision history, purpose, associated documentation, parameter description, assumptions/resources required, side effects, diagnostics, data declarations, packages, operations, and algorithmic code.
- G6-7: Write subprogram bodies to effectively handle interaction with/ effects on their environment.
- G6-8: Write subprogram bodies with one normal exit and a grouped set of abnormal exits via exception handlers.
- G6-9: Write subprogram bodies to pass results back to callers rather than use results to effect their function.
- G6-10: Exploit formal parameter modes to clarify subprogram interface semantics.
- G6-11: Use default parameters to generalize the context of a reusable subprogram; write complete subprogram specifications.
- G6-12: Group all default parameters in subprogram parameter specifications at the end of the specifications.
- G6-13: Use named parameter associations for calls on subprograms with more than three parameters or in any case for interface clarity.

Appendix: List of Guidelines

- G6-14: Minimize subprogram overloading.
- G7-1: Write library unit package specifications and bodies in separate files for ease of recompilation and modification.
- G7-2: Use package specifications to specify the interface to object abstractions; use package bodies to encapsulate implementation-specific details of these abstractions not needed by client software.
- G7-3: Packages should implement interfaces to reusable objects at a consistent abstract level.
- G7-4: Library unit package specifications should have a standard format, including various regions for revision history, purpose, associated documentation, diagnostics, packages, data declarations, operations, and private types.
- G7-5: Secondary unit package bodies should have a standard format including regions for revision history, purpose, associated documentation, assumptions/resources required, side effects, diagnostics, packages, data declarations, operations and initialization code.
- G7-6: Use private or limited private types and the private part of package specifications to restrict client software's view of data and operations on that data.
- G8-1: Do not use "use" context clauses.
- G8-2: Use renaming declarations to resolve name conflicts with the environment.
- G8-3: Use renaming declarations to facilitate modifying reusable software to represent new object abstractions.
- G8-4: Do not hide package standard.
- G9-1: Separate task declarations and bodies for ease of recompilation and modification.
- G9-2: Use task declarations to specify interfaces to reusable objects. Use task bodies to implement these interfaces and properties of the objects.
- G9-3: Write task interfaces at an appropriate abstract level.

A Guidebook for Writing Reusable Source Code in Ada

- G9-4: First-level package-nested task declarations should have a standard format including regions for purpose, entry descriptions, representation clause descriptions, and associated documentation.
- G9-5: Secondary unit (subunit) and first-level package body nested task bodies should have a standard format including regions for revision history, purpose, associated documentation, assumptions/resources required, side effects, diagnostics, packages, data declarations, operations and algorithmic code.
- G9-6: Write task bodies to effectively handle interaction with/effects on their environment; use SHARED variables.
- G9-7: Write task bodies with one normal exit or termination point and a grouped set of abnormal exits via exception handlers.
- G9-8: Write task body accept statements to pass results back to callers of the task rather than using results to effect task function.
- G9-9: Use task types to define reusable operations on data and task objects to implement particular (distinct) instances of these operations.
- G9-10: Exploit entry formal parameter modes to clarify task interface semantics.
- G9-11: Use default entry parameters to generalize the context of a reusable task; write complete task/entry specifications.
- G9-12: Group all default parameters in entry parameter specifications at the end of the specifications.
- G9-13: Use named parameter associations for calls to task entries with greater than three parameters or in any case for interface clarity.
- G9-14: Minimize entry overloading.
- G9-15: Write all select statements with an else part or include a handler for the PROGRAM_ERROR exception at the end of the enclosing task block.
- G9-16: Minimize use of task priorities or modify priorities accordingly when composing tasks with other tasks for the sake of reuse.
- G9-17: Minimize use of abort statements.

Appendix: List of Guidelines

- G10-1: Use library unit package specifications as the encapsulation mechanism for directly reusable software (i.e., data and operations on the data).
- G10-2: Only "first level" nested non-package entities in library unit package specifications form the basis for "catalogued" directly reusable objects/software.
- G10-3: Use secondary unit package bodies, package specifications containing only data, and subunits corresponding to first-level package body nested stubs as the encapsulation mechanism for indirectly reusable software.
- G10-4: "With" clauses on package specifications should reference only data needed in specifications. "With" clauses can be used freely on package bodies as needed.
- G10-5: Use subunits to achieve modularity and ease of recompilation.
- G10-6: Use separate compilation and separate specifications and bodies to achieve modularity and ease of recompilation.
- G10-7: Minimize use of Pragma Elaborate.
- G12-1: Use generic program units (i.e., packages and subprograms) to effectively parameterize reusable software parts.
- G12-2: Use generic program units to precisely specify module interfaces/ imports and exports.
- G12-3: Use generics to allow specification of multiple instances of reusable software as compared to reuse of one shared instance.
- G12-4: Use basetypes rather than subtypes to specify the type of a generic formal object or generic formal subprogram parameter or result types.
- G12-5: Library unit and first-level package nested generic unit declarations should have a standard format, including a region for description of generic parameters as well as standard information required for non-generic subprogram and package declarations.
- G12-6: Separate generic declarations from bodies for ease of recompilation and modification.
- G12-7: Exploit generic formal object parameter modes to clarify interface semantics.
- G12-8: Use generic type definitions to clarify interface semantics and module operation.

A Guidebook for Writing Reusable Source Code in Ada

- G12-9: Use "additional" generic parameters as necessary to effect inheritance of desired operators on generic formal types.
- G12-10: Minimize generic formal subprogram parameter overloading and overloading of subprograms in generic packages.
- G12-11: Minimize use of the box (is <>) notation to specify default generic formal subprograms as parameters.
- G12-12: Use basic operations/attributes associated with generic formal types to provide required generality to generic bodies.
- G12-13: Used named parameter association in actual parameter parts of generic instantiations.
- G12-14: Use default parameters for generic actual parameters whenever possible.
- G12-15: Create particular instantiations of generic units corresponding to common uses of reusable software.

AD-A166 353

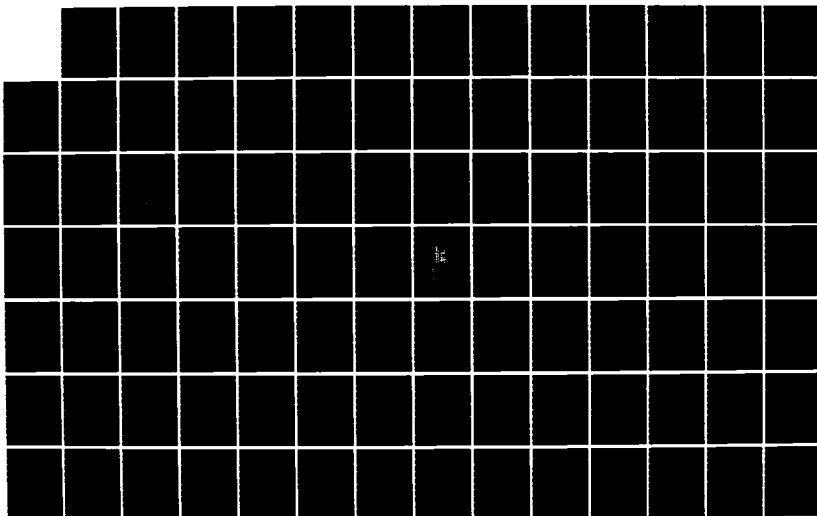
JOINT PROGRAM ON RAPID PROTOTYPING RAPIER (RAPID
PROTOTYPING TO INVESTIGA. (U) HONEYWELL INC GOLDEN
VALLEY MN COMPUTER SCIENCES CENTER 28 MAR 85
N00014-85-C-0666

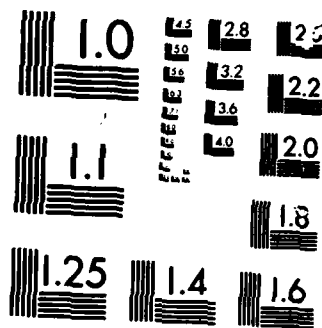
3/4

UNCLASSIFIED

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART

Appendix: Guidebook-Wide Characteristic/Guideline Cross Reference

B APPENDIX: GUIDEBOOK-WIDE CHARACTERISTIC/GUIDELINE CROSS REFERENCE

This appendix contains a cross reference between the reusability characteristics presented in Chapter I-2 and the guidelines presented in Section II.

1. Interface is both syntactically and semantically clear.
G6-3, G6-5, G6-6, G6-10, G6-13, G7-2, G7-4, G7-5, G9-2, G9-4, G9-5, G9-10, G9-13, G12-2, G12-7, G12-8, G12-13.
2. Interface is written at appropriate (abstract) level.
G6-4, G7-3, G9-3, G10-2, G12-2.
3. Component does not interfere with the environment.
G6-7, G8-1, G8-4, G9-6, G9-12, G9-16, G9-17, G10-7.
4. Component is designed as object-oriented; that is, packaged as typed data with procedures and functions which act on that data.
G6-2, G6-3, G7-2, G8-3, G9-2, G9-9, G10-1, G10-3, G12-9.
5. Actions based on function results are made at the next level up.
G6-9, G9-8.
6. Component incorporates scaffolding for use during "building phase."
G6-6, G7-4, G7-5, G9-5, G9-15.
7. Separate the information needed to use software, its specification, from the details of its implementation, its body.
G6-1, G7-1, G9-1, G10-6, G12-6.
8. Component exhibits high cohesion/low coupling.
G6-3, G7-2, G7-6, G9-2, G12-2.

A Guidebook for Writing Reusable Source Code in Ada

9. Component and interface are written to be readable by persons other than the author.

G6-5, G6-6, G6-8, G6-13, G6-14, G7-4, G7-5, G8-2, G8-4, G9-4, G9-5, G9-7, G9-13, G9-14, G10-5, G12-1, G12-4, G12-5, G12-10, G12-13, G12-14.

10. Component is written with the right balance between generality and specificity.

G6-11, G9-11, G12-1, G12-12, G12-14.

11. Component is accompanied by sufficient documentation to make it findable.

G6-5, G6-6, G7-4, G7-5, G9-4, G9-5, G12-5.

12. Component can be used without change or with only minor modifications.

G6-1, G6-3, G6-8, G6-12, G7-1, G7-2, G8-1, G8-3, G9-1, G9-2, G9-7, G9-12, G10-5, G10-6, G12-1, G12-6.

13. Insulate a component from host/target dependencies and assumptions about its environment; Isolate a component from format and content of information passed through it which it does not use.

G6-3, G6-14, G7-2, G7-6, G8-2, G9-2, G9-14, G10-4, G10-7, G12-2, G12-10, G12-11.

14. Component is standardized in the areas of invoking, controlling, terminating its function, error-handling, communication, and structure.

G6-5, G6-6, G6-8, G6-12, G7-4, G7-5, G9-4, G9-5, G9-7, G9-12, G10-1, G10-3, G12-5.

15. Components should be written to exploit domain of applicability. Components should constitute the right abstraction and modularity for the application.

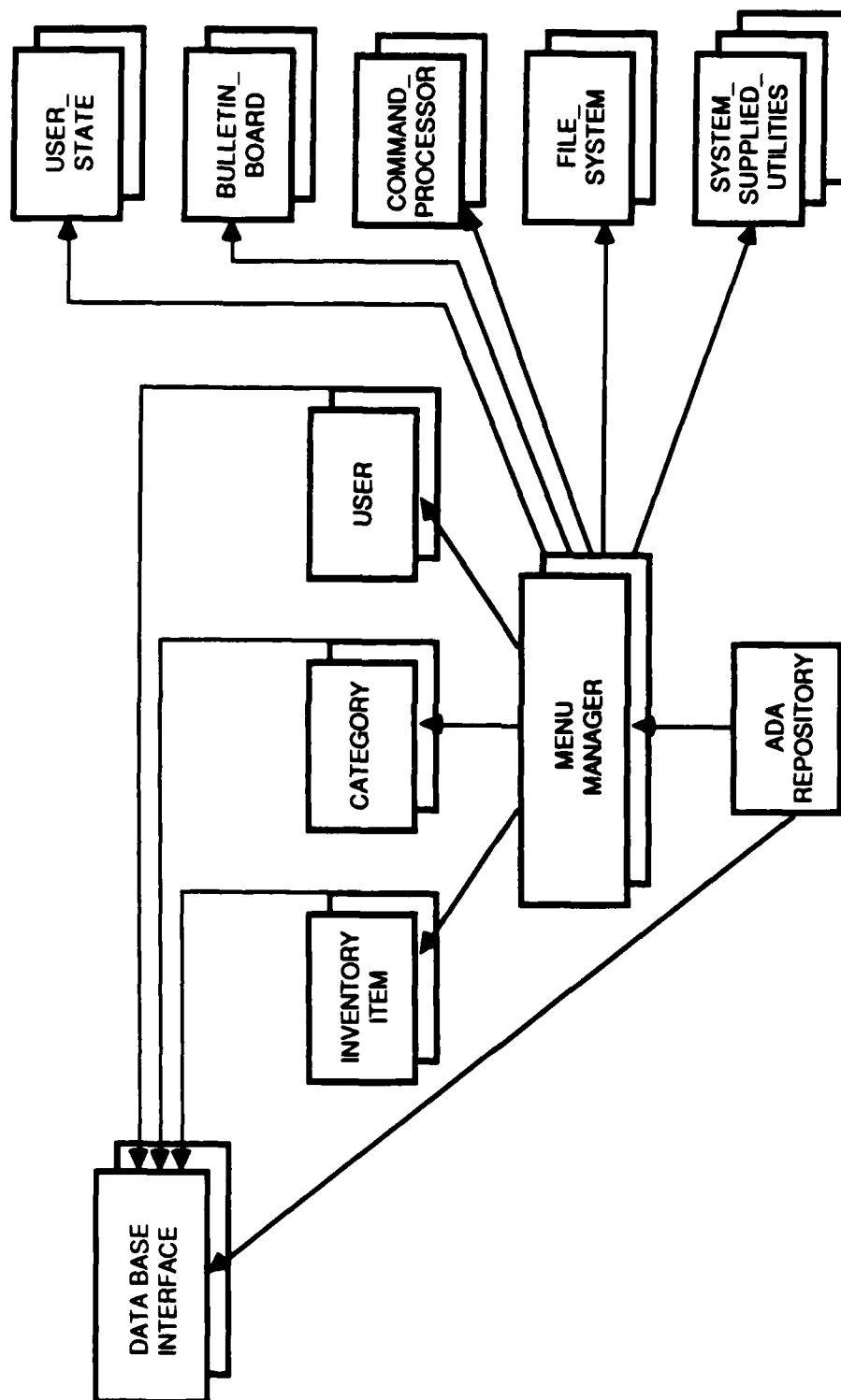
G12-3, G12-15.

C APPENDIX: EXAMPLE ADA MODULES

C.1 DESCRIPTION OF EXAMPLE MODULES

The example Ada modules contained in this appendix are taken from the design of a reusable software repository developed at the Honeywell Computer Sciences Center. This repository supports retrieval, submission, and maintenance of categories of inventory items stored in a database management system. Its user interface is menu oriented. Figure C-1 shows the major components of the repository:

- Ada_Repository : a procedure that is the main routine;
- Menu_Manager : a generic package specification and body defining repository menu objects and operations on them; contains operations including task LOG_REPOSITORY_USE and procedure CREATE_INITIAL_MENU;
- Inventory_Item : a package specification and body defining repository inventory items and operations on them;
- Category : a package specification and body defining repository categories and operations on them;
- User : a package specification and body defining repository user objects and operations on them;
- Database_Interface : a package specification and body containing the interface to the underlying database management system (entities, attributes, and relationships);
- User_State : a package specification and body defining user state information and associated operations;
- Bulletin_Board : a package specification and body defining a repository bulletin board and associated operations;
- Command_Processor : a package specification and body providing the ability to access and execute host commands;
- File_System : a package specification and body providing access to host files;
- System_Supplied_Uilities : a package specification and body providing system supplied utilities, including STRING manipulation.



File No. 6-0366

Figure C-1 Ada Repository Example

The example Ada modules below make up a subset of the repository components. These modules are: (1) procedure `Ada_Repository`, (2) generic package `Menu_Manager` (specification), (3) generic package `Menu_Manager` (body), (4) procedure body `Create_Initial_Menu` (subunit), (5) task body `Log_Repository_Use` (subunit), and (6) package `Inventory_Item` (specification).

C.2 GUIDELINES ILLUSTRATED

We have attempted to illustrate as many reusability guidelines as possible with the example Ada modules below. Most notably, the following guidelines (grouped by reusability characteristic) are illustrated:

1. Interface is both syntactically and semantically clear.

G6-3, G6-5, G6-6, G6-10, G7-2, G7-4, G7-5, G9-2, G9-4, G9-5, G9-10, G12-2, G12-8, G12-13.

4. Component is designed as object-oriented; that is, packaged as typed data with procedures and functions which act on that data.

G6-2, G6-3, G7-2, G9-2, G9-9, G10-1, G10-3.

6. Component incorporates scaffolding for use during "building phase."

G6-6, G7-4, G7-5, G9-5.

7. Separate the information needed to use software, its specification, from the details of its implementation, its body.

G6-1, G7-1, G9-1, G10-6, G12-6.

9. Component and interface are written to be readable by persons other than the author.

G6-5, G6-6, G6-8, G6-14, G7-4, G7-5, G8-4, G9-4, G9-5, G9-7, G9-14, G10-5, G12-1, G12-4, G12-5, G12-10, G12-13.

11. Component is accompanied by sufficient documentation to make it findable.

G6-5, G6-6, G7-4, G7-5, G9-4, G9-5, G12-5.

14. Component is standardized in the areas of invoking, controlling, terminating its function, error-handling, communication, and structure.

A Guidebook for Writing Reusable Source Code in Ada

G6-5, G6-6, G6-8, G7-4, G7-5, G9-4, G9-5, G9-7, G10-1, G10-3, G12-5.

Note: Procedure `Ada_Repository` is "officially" not reusable since it is the main routine and is not enclosed in a library unit package. However, we have written it using our template for reusable subprograms.

C.3 ADA MODULES

```
*****
*                               EXAMPLE:  MODULE 1                               *
*****

with TEXT_IO;
with MENU_MANAGER, DATABASE_INTERFACE;
procedure ADA_REPOSITORY is

-- Revision History:  Created 2/23/86 R. St. Dennis
-- Purpose:
--   Explanation: Main routine for a repository of Ada inventory items.
--               This repository supports retrieval, submission, and
--               maintenance functions.
--   Keywords:   None

-- Associated Documentation: Design for Honeywell Reusable Software Repository
-- Parameter Description: None
-- Assumptions/Resources Required: None
-- Side Effects: None
-- Diagnostics: None
-- Data Declarations:
--   Types:
--     type AR_NUMBER is range 1..100;
--     type AR_MENU_ITEM is range 1..55;
--     type AR_MENU is array (AR_MENU_ITEM) of STRING;
--   Objects:
--     INITIAL_MENU_NUMBER : AR_NUMBER;
--     INITIAL_MENU_ITEM   : AR_MENU_ITEM;
--     CONTINUE            : BOOLEAN;
--     LOG_DESIGNATOR      : STRING;
-- Packages:
--   package REPOSITORY_MENU_MANAGER is new MENU_MANAGER (MENU => AR_MENU,
--                                                         MENU_NUMBER => AR_NUMBER,
--                                                         MENU_ITEM => AR_MENU_ITEM);

-- Operations:
--   Subprograms: None
--   Tasks: None
--   Algorithm:
```



```

begin -- ADA_REPOSITORY

  DATABASE_INTERFACE.INITIALIZE;
  REPOSITORY_MENU_MANAGER.CREATE_INITIAL_MENU(INITIAL_MENU_NUMBER);
  REPOSITORY_MENU_MANAGER.LOG_REPOSITORY_USE.OPEN_LOG(LOG_DESIGNATOR);

  while CONTINUE loop
    REPOSITORY_MENU_MANAGER.DISPLAY_MENU(INITIAL_MENU_NUMBER);
    REPOSITORY_MENU_MANAGER.ACCEPT_MENU_RESPONSE(INITIAL_MENU_NUMBER,
                                                    INITIAL_MENU_ITEM);
    REPOSITORY_MENU_MANAGER.PROCESS_MENU_RESPONSE(INITIAL_MENU_NUMBER,
                                                    INITIAL_MENU_ITEM,
                                                    CONTINUE);
  end loop;

  REPOSITORY_MENU_MANAGER.LOG_REPOSITORY_USE.CLOSE_LOG(LOG_DESIGNATOR);
  DATABASE_INTERFACE.FINALIZE;

  exception

    when REPOSITORY_MENU_MANAGER.MENU_MANAGEMENT_ERROR =>
      REPOSITORY_MENU_MANAGER.LOG_REPOSITORY_USE.CLOSE_LOG(LOG_DESIGNATOR);
      DATABASE_INTERFACE.FINALIZE;
      TEXT_IO.PUT("Repository execution terminated due to MENU_MANAGEMENT
                  error");

    when others=>
      REPOSITORY_MENU_MANAGER.LOG_REPOSITORY_USE.CLOSE_LOG(LOG_DESIGNATOR);
      DATABASE_INTERFACE.FINALIZE;
      -- Report termination of repository execution.

end ADA_REPOSITORY;

```

```

*****
*                               EXAMPLE:  MODULE 2                               *
*****

```

```

with DATABASE_INTERFACE;
generic

```

```

  type MENU is private;
  type MENU_NUMBER is range <>;
  type MENU_ITEM is range <>;

```

```

package MENU_MANAGER is

```

```

  -- Revision History: Created 2/20/86  R. St. Dennis

```

A Guidebook for Writing Reusable Source Code in Ada

```
-- Purpose:
--   Explanation: Provide data structures for and operations on
--                 repository menu objects.
--   Keywords:    menu, menu_manager

-- Generic Parameter Description:
--   MENU         : Specific menu type desired; must be an array of STRINGS.
--   MENU_NUMBER  : Integer type for number of menus.
--   MENU_ITEM    : Integer type for range of items within menus.

-- Associated Documentation: Design for Honeywell Reusable Software Repository

-- Diagnostics:
--   MENU_MANAGEMENT_ERROR : exception;

-- Packages: None

-- Data Declarations:
--   Types:   None
--   Objects: None

-- Operations:
--   Subprograms:
```

```
procedure CREATE_INITIAL_MENU (M_NUMBER : out MENU_NUMBER);
```

```
-- Purpose:
--   Explanation: Create initial repository menu.
--   Keywords:    initial_menu, create_initial_menu

-- Parameter Description:
--   M_NUMBER : Menu number associated with initial menu.

-- Associated Documentation: same as above
```

```
procedure CREATE_CATEGORY_MENU (CATEGORY      : in
                                DATABASE_INTERFACE.RELATION_NAME;
                                CATEGORY_MENU : out MENU);
```

```
-- Purpose:
--   Explanation: Create menu of categories from repository contents.
--   Keywords:    category_menu, create_category_menu

-- Parameter Description:
--   CATEGORY      : Parent category name.
--   CATEGORY_MENU : Category menu created.

-- Associated Documentation : same as above
```

```

-----
procedure DISPLAY_MENU (M_NUMBER : in MENU_NUMBER);

-- Purpose:
--   Explanation: Displays specific menu.
--   Keywords:    display_menu

-- Parameter Description:
--   M_NUMBER : Number of menu.

-- Associated Documentation : same as above

-----

procedure ACCEPT_MENU_RESPONSE (M_NUMBER      : in MENU_NUMBER;
                                MENU_ITEM_SELECTED : out MENU_ITEM);

-- Purpose:
--   Explanation: Accept menu response.
--   Keywords:    menu_response, accept_menu_response

-- Parameter Description:
--   M_NUMBER      : Number of menu.
--   MENU_ITEM_SELECTED : Specific item from menu selected.

-- Associated Documentation: same as above

-----

procedure PROCESS_MENU_RESPONSE (M_NUMBER      : in MENU_NUMBER;
                                MENU_ITEM_SELECTED : in MENU_ITEM;
                                EXIT              : out BOOLEAN);

-- Purpose:
--   Explanation: Process response specified by menu selection.
--               This processing may involve a call to DISPLAY_MENU and
--               ACCEPT_MENU_RESPONSE and a recursive call to PROCESS_
--               MENU_RESPONSE.
--   Keywords:    menu_response, process_menu_response.

-- Parameter Description:
--   M_NUMBER      : Number of menu.
--   MENU_ITEM_SELECTED : Specific item from menu selected.
--   EXIT          : Indication to exit menu system.

-- Associated Documentation: same as above

-----

-- Tasks:
task LOG_REPOSITORY_USE is

-- Purpose:
--   Explanation: Record usage of repository.
--   Keywords:    log, log_repository_use.

```

A Guidebook for Writing Reusable Source Code in Ada

```
-- Entries:
  entry OPEN_LOG (LOG_ID : out STRING);  -- Open repository log named
                                         -- LOG_ID;
  ...

  entry CLOSE_LOG (LOG_ID : in STRING);  -- Close repository log named
                                         -- LOG_ID;

-- Associated Documentation:  same as above

end LOG_REPOSITORY_USE;

-----

-- Private:

  private

  ...

end MENU_MANAGER;
```

```
*****
*                                     *
*               EXAMPLE:  MODULE 3   *
*                                     *
*****
```

```
with INVENTORY_ITEM, CATEGORY, USER, TEXT_IO;
with USER_STATE, BULLETIN_BOARD, COMMAND_PROCESSOR, FILE_SYSTEM,
  SYSTEM_SUPPLIED_UTILITIES;
package body MENU_MANAGER is

  -- Revision History: Created 02/21/86  R. St.Dennis
  -- Purpose:
  --   Explanation: Provide data structures for and operations on
  --                 repository menu objects.
  --   Keywords:    menu, menu_manager

  -- Associated Documentation: Design for Honeywell Reusable Software
  --                           Repository.
  -- Assumptions/Resources Required: None
  -- Side Effects: None
  -- Diagnostics: None
  -- Packages: None
  -- Data Declarations:
  --   Types:
  type MENU_ACCESS is access MENU;
  type MENU_STACK_ELEMENT is
    record
      MENU_POINTER          : MENU_ACCESS;
      MENU_FILESYS_LOCATION : STRING (1..100);
```

```

        end record;
--   Objects:
    MENU_STACK      : array (1..31) of MENU_STACK_ELEMENT;
    MENU_STACK_INDEX : NATURAL := 0;
--   Operations:
--   Subprograms:
    procedure CREATE_INITIAL_MENU (M_NUMBER : out MENU_NUMBER)
                                   is separate;
    procedure CREATE_CATEGORY_MENU (CATEGORY      : in
                                     DATABASE_INTERFACE.RELATION_NAME;
                                     CATEGORY_MENU : out MENU) is separate;
    procedure DISPLAY_MENU (M_NUMBER : in MENU_NUMBER) is separate;
    procedure ACCEPT_MENU_RESPONSE (M_NUMBER      : in MENU_NUMBER;
                                    MENU_ITEM_SELECTED : out MENU_ITEM)
                                    is separate;
    procedure PROCESS_MENU_RESPONSE (M_NUMBER      : in MENU_NUMBER;
                                    MENU_ITEM_SELECTED : in MENU_ITEM;
                                    EXIT              : out BOOLEAN)
                                    is separate;
--   Tasks:
    task body LOG_REPOSITORY_USE is separate;
--   Initialization:

begin
exception
    when IVENTORY_ITEM.INVENTORY_ITEM_ERROR =>
        . . .

        raise MENU_MANAGEMENT_ERROR;

    when others =>

        . . .

        raise MENU_MANAGEMENT_ERROR;

end MENU_MANAGER;

```

```

*****
*                               EXAMPLE:  MODULE 4                               *
*****

```

```

separate (MENU_MANAGER)
procedure CREATE_INITIAL_MENU (M_NUMBER : out MENU_NUMBER) is

```

```

-- Revision History:  Created 2/21/86  R. St. Dennis
-- Purpose:
--   Explanation:  Creates initial repository menu by reading data for it from
--                 a host file and placing it on the MENU_MANAGER menu stack.
--   Keywords:    initial_menu, create_initial_menu

```

A Guidebook for Writing Reusable Source Code in Ada

```
-- Associated Documentation: Design for Honeywell Reusable Software Repository
-- Parameter Description:
-- M_NUMBER : Number of menu created.
```

```
-- Assumptions/Resources Required: None
-- Side Effects: None
-- Diagnostics: None
-- Packages: None
-- Data Declarations:
--   Types: None
--   Objects:
```

```
    FILE_DESIGNATOR : FILE_SYSTEM.FILE_NAME := "DRAO:[SOURCE]FILE_NAME.TXT";
```

```
-- Operations:
--   Subprograms: None
--   Tasks: None
-- Algorithm:
```

```
begin -- CREATE_INITIAL_MENU
```

```
    -- read from host file, create menu, and place on MENU_STACK;
    -- increment MENU_STACK_INDEX by 1;
```

```
    M_NUMBER := MENU_STACK_INDEX + 1;
```

```
    exception
```

```
        when others =>
```

```
        ...
```

```
end CREATE_INITIAL_MENU;
```

```
*****
*                                     EXAMPLE: MODULE 5                                     *
*****
```

```
separate (MENU_MANAGER)
task body LOG_REPOSITORY_USE is
```

```
    -- Revision History: Created 02/21/86 R. St. Dennis
    -- Purpose:
    --   Explanation: Record usage of repository.
    --   Keywords:    log, log_repository_use
```

```
    -- Associated Documentation: Design for Honeywell Reusable Software
    --                               Repository
    -- Assumptions/Resources Required: None
    -- Side Effects: None
```

```

-- Diagnostics: None
-- Packages: None
-- Data Declarations:
--   Types: None
--   Objects: None
-- Operations:
--   Subprograms: None
--   Tasks: None
-- Algorithm:

begin
  .
  .
  .
  accept OPEN_LOG (LOG_DESIGNATOR : out STRING) do
    .
    TEXT_IO.GET(LOG_DESIGNATOR);
    .
  end OPEN_LOG;

  accept CLOSE_LOG (LOG_DESIGNATOR : in STRING) do
    .
    .
    .
  end CLOSE_LOG;
  exception
    when others =>
      . . .

end LOG_REPOSITORY_USE;

```

```

*****
*                               *
*               EXAMPLE: MODULE 6               *
*                               *
*****

```

```

package INVENTORY_ITEM is

```

```

  -- Revision History:   Created 02/21/86  R. St. Dennis
  -- Purpose:
  --   Explanation: Provide data structures for and operations on
  --                 repository inventory item objects.
  --   Keywords:      inventory_item

  -- Associated Documentation: Design for Honeywell Reusable
  --                           Software Repository.

  -- Diagnostics:
  --   IVENTORY_ITEM_ERROR : exception;
  -- Packages: None
  -- Data Declarations:
  --   Types:

```

```
type INVENTORY_ITEM_INDEX is new POSITIVE;
-- Objects: None
-- Operations:
-- Subprograms:
-----
procedure DISPLAY_ACTUAL_INVENTORY_ITEM (II_NUMBER : in
                                         INVENTORY_ITEM_INDEX);

-- Purpose:
-- Explanation: Display actual inventory item on terminal
--              screen.
-- Keywords:    display, display_actual_inventory_item

-- Parameter Description:
-- II_NUMBER : Number associated with inventory item in
--              repository

-- Associated Documentation : same as above
-----
procedure COPY_INVENTORY_ITEM (II_NUMBER : in
                               INVENTORY_ITEM_INDEX);

-- Purpose:
-- Explanation: Copy inventory item to host file in current
--              working directory.
-- Keywords:    copy, copy_inventory_item

-- Parameter Description:
-- II_NUMBER : Number associated with inventory item in
--              repository

-- Associated Documentation : same as above
-----
procedure DISPLAY_INVENTORY_ITEM_BASIC_INFO (II_NUMBER : in
                                              INVENTORY_ITEM_INDEX);

-- Purpose:
-- Explanation: Display basic information about inventory
--              item including author, date submitted, number
--              of users, etc.
-- Keywords:    display, display_inventory, item_basic_info

-- Parameter Description:
-- II_NUMBER : Number associated with inventory item in
--              repository

-- Associated Documentation : same as above
```


Appendix: Example Ada Modules

```
-----
procedure DISPLAY_INVENTORY_ITEM_DEFICIENCIES (II_NUMBER : in
                                                INVENTORY_ITEM_INDEX);

-- Purpose:
--   Explanation: Display list of deficiencies associated with
--                 inventory item.
--   Keywords:    display, display_deficiencies, deficiencies

-- Parameter Description:
--   II_NUMBER : Number associated with inventory item in
--                 repository

-- Associated Documentation : same as above

-----
--   Tasks:
--   Private:

    private
    . . .

end INVENTORY_ITEM;
```

D APPENDIX: GLOSSARY

cohesion -

The type of association among the component elements of a module. Functional cohesion is that type in which every element within a module contributes directly to performing one single function.

client software -

(In this guidebook) software that references reusable Ada software.

coupling -

A measure of the strength of interconnection (the communication bandwidth) between modules. High coupling results from low cohesion.

directly reusable software parts -

Software parts whose behavior or effect is cataloged, that is, "advertised" in a catalog of reusable software that developers use to determine what software parts are available for reuse. A catalog can be an automated software repository's classification scheme, a list of component names and descriptions on paper, or even a rumor the developer hears from a colleague down the hall. Directly reusable software parts are what developers search for and choose.

indirectly reusable software parts -

Software parts that support directly reusable parts. Indirectly reusable software parts provide the environment, the ancillary definitions and data that directly reusable parts need in order to perform correctly. In the ideal case, indirectly reusable parts are incorporated into programs under construction automatically by a software base management system.

users -

(In this guidebook), humans who reuse Ada source code.

BIBLIOGRAPHY

[[RAPIER86]]

RaPIER Project. "Final Scientific Report: RaPIER Project (Contract No. N00014-85-C-0666)," Honeywell Computer Sciences Center, Golden Valley, MN, March 1986.

[AMANO84]

K. Amano, M. Chiba, A. Mochida and T. Maeda. "An Approach Toward Integrated Algorithm Information Systems," Information Systems, Vol. 9 No. 3/4, 1984.

[BENTLEY85]

Jon Bentley. "Programming Pearls," Communications of the ACM, Vol. 28 No. 7, July 1985, pp. 671-679.

[BERGLAND81]

G. D. Bergland. "A Guided Tour of Program Design Methodologies," IEEE Computer, Vol. 14 No. 10, October 1981, pp. 13-37.

[BIENIAK85]

R. M. Bieniak, L. M. Griffin, L. R. Tripp. "Position Paper: Automated Parts Composition," Proceedings of STARS Reusability Workshop, April 1985.

[BIGGERSTAFF84]

Ted J. Biggerstaff, Alan J. Perlis. "Forward: Special Issue on Software Reusability," IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 474-477.

[BOOCH83]

Grady Booch. "Object-oriented Design," Tutorial on Software Design Techniques. Ed. P. Freeman and A. Wasserman, 4th edition (Catalog Number EH0205-5), IEEE Computer Society Press, 1983.

[BOOCH83a]

Grady Booch. Software Engineering with Ada, The Benjamin/Cummings Publishing Company, Inc., 1983.

Bibliography

- [BOOCH83b]
Grady Booch. "Dear Ada," Ada Letters, Vol. III, No. 3, ACM SIGAda, November/December 1983, pp. 25-28.
- [BOOCH85]
Grady Booch. "ACM SIGAda Tutorial: Ada Methodologies," ACM SIGAda Meeting, July 30, 1985.
- [DOD83]
United States Department of Defense. Reference Manual for the Ada Language: ANSI/MIL-STD-1815A, United States Department of Defense, January 1983.
- [FRANKOWSKI85b]
Elaine N. Frankowski, Christine M. Anderson. "Design/Integration Panel Report," Proceedings of the STARS Reusability Workshop, April 1985.
- [FREEMAN83]
Peter Freeman, Anthony I. Wasserman. "Introduction to Part III: Specification Methods," in Tutorial on Software Design Techniques, IEEE Computer Society, August 1983, pp. 173-176.
- [GOGUEN84]
J. A. Goguen. "Parameterized Programming," IEEE Transactions on Software Engineering, Vol. SE-10 No. 5, September 1984, pp. 528-543.
- [GOLDBERG83]
Adele Goldberg, D. Robson. SMALLTALK-80: The Language and Its Implementation, Addison-Wesley, Reading, MA, 1983.
- [HARRIS85]
Harris Corporation. "Draft GKS Binding to ANSI Ada," Harris Corporation, Melbourne, FL, February 1, 1985.
- [HOROWITZ84]
Ellis Horowitz, John B. Munson. "An Expansive View of Reusable Software," IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 477-487.

A Guidebook for Writing Reusable Source Code in Ada

[ISSI86]

International Software Systems, Inc.. "PSDL: Prototype System Description Language," ISSI Technical Report, unnumbered, January 30, 1986.

[JONES84]

T. Capers Jones. "Reusability in Programming: A Survey of the State of the Art," IEEE Transactions on Software Engineering, Vol. SE-10 No. 5, September 1984, pp. 488-494.

[KERNIGHAN84]

Brian W. Kernighan. "The Unix System and Software Reusability," IEEE Transactions on Software Engineering, Vol. SE-10 No. 5, September 1984, pp. 513-518.

[LANERGAN84]

Robert G. Lanergan, Charles A. Grasso. "Software Engineering with Reusable Designs and Code," IEEE Trans. on Software Engineering, Vol. SE-10D, No. 5, September 1984.

[MacLENNAN85]

Bruce J. MacLennan. "A Simple Software Environment Based on Objects and Relations," Naval Postgraduate School Technical Report, NPS52-85-005, April 1985.

[MATSUMOTO84]

Yoshihiro Matsumoto. "Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels," IEEE Transactions on Software Engineering, Vol. SE-10 No. 5, September 1984, pp. 502-513.

[NEIGHBORS84]

James M. Neighbors. "The Draco Approach to Constructing Software from Reusable Components," IEEE Transactions on Software Engineering, Vol. SE-10 No. 5, September 1984, pp. 564-574.

[STANDISH84]

Thomas A. Standish. "An Essay on Software Reuse," IEEE Transactions on Software Engineering, Vol. SE-10 No. 5, September 1984, pp. 494-497.

[STARS83]

Stars. Software Technology for Adaptable, Reliable Systems (STARS) Program Strategy, U. S. Department of Defense, April 1983.

[SYMBOLICS84]

Symbolics Inc.. "Lisp Flavors," Symbolics 3600 Series User's Manual, Vol. 3B, Symbolics Inc, Cambridge, MA, 1984.

[WEBSTER77]

Merriam Co.. Webster's New Collegiate Dictionary, G. & C. Merriam Co., Springfield, MA, 1977, pp. 990.

[YEH84]

Raymond T. Yeh, Nicholas Roussopoulos. "Management of Reusable Software," IEEE Compcon 84, Arlington, Virginia, September 16-20, 1984, pp. 311-320.

6.4 FUTURE WORK

In the upcoming year we plan to complete the remaining sections of the Reusability Guidebook and refine/add to the Ada examples it provides. Guidelines from the Guidebook will be used to construct reusable software components for RaPIER's software base management system. We also plan to circulate the Guidebook for review. Feedback we receive from RaPIER prototyping experiments and the Guidebook review will enable us to evaluate the characteristics and guidelines and refine them accordingly.

In later years, we hope to develop measures for the Reusability Guidebook characteristics and implement/experiment with software base classification schemes tailored to Guidebook characteristics and guidelines.

blank back page

SECTION 7

FRAGMENT GENERATION STUDY

This section describes initial work on Fragment Generation which was accomplished solely with Honeywell internal funds. The task was to study current fragment generation methods, determine their applicability to RaPIER, and recommend a plan for absorbing applicable technology into the RaPIER environment. If found applicable, the study should recommend a program plan.

7.1 PROBLEM STATEMENT

The RaPIER (Rapid Prototyping to Identify End-user Requirements) project will deliver methodologies and automated support for constructing prototypes rapidly and cheaply. The prototypes can be constructed through a combination of the following:

- o Reuse of software components. The components are program modules that are catalogued and managed by a Software Base. Software Base components are, in general, small units of code that can be composed into larger components using a Very High Level Language (VHLL).
- o Reuse of larger fragments. Large pieces of code such as text-editors or database managers should be reused with little or no alteration.
- o Fragment generation. Code that does not exist must be either written in a conventional programming language (assembler or high-level language) or generated from a high-level specification by program transformation.

So far, progress has been made in the reuse of software components and larger fragments. A Software Base is being built and we have reused such large fragments as the EMACS text-editor, the MULTICS Compose text-processor and various data managers without significantly altering their code. The third option, fragment generation is being investigated. Specifically, we prefer the economically attractive method of generating program fragments by transforming high-level specifications into executable code.

7.2 OUTCOME

We conducted a literature survey of current program transformation methods. Given the state-of-the-technology and given the resources available to RaPIER for the duration of the project, we made the recommendations in subsection 7.4. This document is neither a tutorial nor a survey but an evaluation as to whether the RaPIER project can use today's program transformation technologies or add value to make them useful in our environment.

7.3 PROGRAM TRANSFORMATION SYSTEMS

By program transformations, we mean the systematic transformation of a formal specification into executable code through the application of correctness-preserving rules known as transformation rules. A typical example is the generation of a language parser: a formal specification (usually a BNF grammar) is fed into the parser generator which in turn generates executable code for parsing the language. A detailed survey of transformation systems can be found in [PATCH83]. In this section, we will discuss three systems which represent the state-of-the-technology.

A. The DRACO-System

The DRACO System [NEIGHBORS80] provides a programming environment in which the design and analysis of programs are reused. DRACO provides mechanisms for defining domain-specific specification languages, appropriate mappings from those languages into any of a number of executable high-level languages or other previously defined specification languages, and optimizing rules. The transformation from the domain language into the target language is semi-automatic in the sense that the user can make individual implementation choices (called refinements in DRACO) or even insert new tactics into the system when "guiding" the transformation.

EXAMPLE

The following example is taken from [NEIGHBORS80]. The problem is to generate transactions against a given relational database from natural language queries. The domains involved here are the domain of augmented transition networks (ATN) and the domain of relational database management systems. A similar problem in which natural language parsers were generated given a dictionary and an ATN was the motivation for this particular problem's solution. In the previous problem, the dictionary specified the legal words, their parts of speech, and special word features, while the ATN is a finite state machine implementation of rules for combining words into correct sentences. The input to the ATN is a dictionary and a sentence; the output is a set of syntax trees.

For the new example, the dictionary specifies the domain in which the database operates - the schema along with words for constructing sentences. The database objects are nouns in the dictionary, verbs imply a relation, and so forth, as shown in Figure 7-1.

```

/*
 * NOUN = noun may imply a restriction
 * NPR = indicates nominative pronoun
 * NUM = number of the noun
 * TYPE = indicates a restriction
 * ROOT = gives the type restriction
 */
Fred | NOUN | NPR NUM : Singular
Ethel | NOUN | NPR NUM : Singular
.
.
.
/*
 * VERB = verb implies a relation
 * NUM = number of the verb
 * REL = verb relation name
 * SDOM = subject domain in relation
 * ODOM = object domain in relation
 */
IS | VERB | NUM: Singular | SDOM: OBJ | ODOM: Type | REL: IS
ARE | VERB | NUM: Plural | SDOM: OBJ | ODOM: Type | REL: IS
.
.
.
/* DETERMINERS */
.
.
.
/* ADJECTIVES */
.
.
.
/* COMMANDS */
find | CMD |
what | CMD |
.
.
.
/* QUANTIFIERS */
.
.
.

```

Figure 7-1 Dictionary_Block

The ATN for this problem was modified to build nested transactions instead of syntax trees.

There are other examples in which the user is involved in the refinement process. For this class of transformations, DRACO reuses designs and specifications already stored in the system. By interrogating the user, DRACO is helped to choose among various implementation alternatives and then refines the chosen alternative into executable code.

B. The TAMPR System

The TAMPR (Transformation Assisted Multiple Program Realization) System [BOYLE79] is one of the oldest program generation systems. It has been in use at the Applied Mathematics Division of the Argonne National Laboratory to assist the development, adaptation, and maintenance of mathematical software packages written in FORTRAN. As in the DRACO System, an abstract program is transformed into executable code. The emphasis in TAMPR is on implementations that are also optimized for the hardware/software environments or for the problem domain.

EXAMPLE

This example is taken from [BOYLE79]. It illustrates the transformation of high-level specification into code through the replacement of patterns by program text; code optimization is also done along with the replacement (called macro-expansion in TAMPR). Below are transformations for implementing the push and pop subroutines.

```

<stmt>
{.SD
  call push (<expr> "1")
-->
  push: block;
    sp = sp + 1;
    if (sp .GT. maxsp) then;
      call error;
    end;
    stack (sp) = <expr> "1";
  push : end
.SC.

.SD.
  call pop (<var> "1")
-->
  pop : block;
    if (sp .LE. 0) then;
      call error;
    end;
    <var> "1" = stack (sp);
    sp = sp - 1;
  pop : end;
.SC }

```

The transformation process is straightforward except that there is no optimization. The optimization process is carried out by introducing extra constructs into the transformation rules. For a specification such as:

```

.
.
call pop (symbol);
<stmt>
.
.
<stmt>
call push (identifier);
.
.

```

The unoptimized result is:

```
.
.
pop : block;
      if (sp .LE. 0) then;
        call error;
      end
      symbol = stack (sp);
      sp    = sp - 1;
pop : end;
<stmt>;
.
.
<stmt>;
push : block;
      sp = sp + 1;
      if (sp .GT. maxsp) then;
        call error;
      end
      stack (sp) = identifier;
push : end;
<stmt>
.
.
```

Let us consider one possible optimization. If push follows a pop, and if no other push or pop operations are in between them, then the test $(sp > maxsp)$ can be removed from the push operation because the space for the element to be pushed is certainly available following the pop. One way to detect that there is no push or pop operation in between a pop and push sequence is to bundle all non-stack operations into one block so that stack operations are conspicuous. The example specification can then be written as:

```
.
.
pop : begin ; <stmt tail>;
no stack operations ; begin <stmt tail>;
push : begin ; <stmt tail>;
<stmt tail>
.
.
```

Such clauses as ".where. (condition)" can also be used to guide certain optimizations. The "where" clause is similar to an "if" statement in programming languages; in this case, the transformation system evaluates the condition and acts accordingly.

C. The GIST System

GIST is the name of a high-level specification language developed at the University of Southern California [FEATHER83]. GIST specifications are transformed into code by a process in which the GIST system interacts with

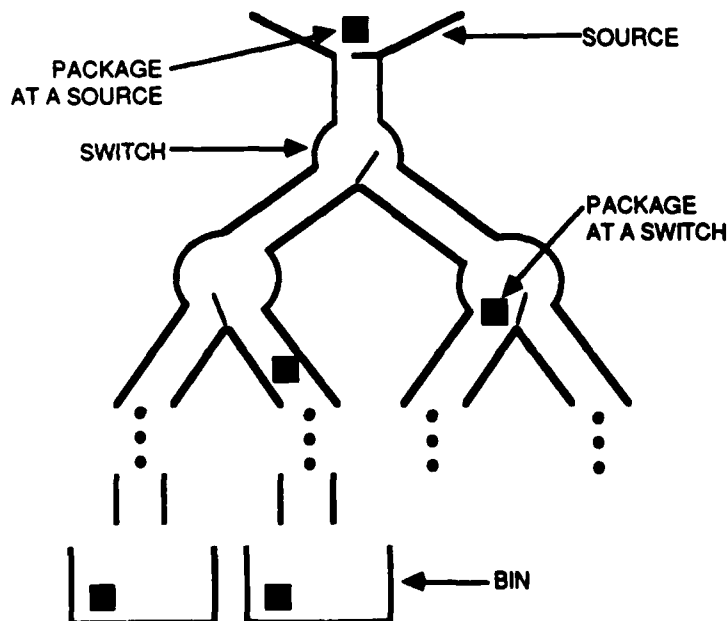
the user. The system is semi-automatic. GIST specifications attempt to formalize natural language descriptions in ways that retain most of the power of natural language. The capabilities of GIST include:

- A relational model of data for associative retrieval of data objects.
- Schemas for describing object relationships.
- Ability to make historical references to past states of objects.
- Constraints which can be declared globally.
- Demons which can be triggered.
- Closed system specification, i.e., once interfaces and global behavior are defined, the behavior of portions of the same become defined implicitly.

EXAMPLE

The following example is taken from [FEATHER83].

A package router (see Figure 7-2 and Figure 2 on page 50 of [FEATHER83]) can be characterized as a network with a source, pipes that connect this source to switches or that connect switches to other switches. The whole connection of switches and pipes is a binary tree. At the end (leaves) are bins which are the destination of packages.



File No. 6-0349

Figure 7-2 Package Router Example

GIST capabilities come in handy here. The network, packages, switches, pipes and bins are all objects whose descriptions and relationships are modelled by database relations. The states of these objects (for example the arrival of a package) can be modelled by insertions, deletions and modification of the relations. The relational approach also allows descriptive references such as: "the bin that is the destination of this package". A given specification can be reused by simply inserting new objects and modifying the schema. Historical references can be made by interrogating the past states of the database; for example "Has this package been at that switch?" Constraints and demons can be modelled in the same way as they are declared against a relational database.

The mapping of GIST constructs into programs is accomplished by converting the relational schema into appropriate data structures (for example arrays, hash tables), generating the appropriate inference rules on the data structures, generating exception handlers for the demon constructs, and so forth. The details of these mappings will not be discussed here. GIST designers insist that the mappings are semi-automatic and there is no immediate plan for full automation.

7.4 DISCUSSION

The designers of the transformation systems discussed in this section agree that productivity gains and correct maintenance of software can be achieved by reusing and modifying high-level specifications rather than code. Each of the systems has been successful in some way. DRACO can store high-level specifications, reuse them, and transform them into code. The problems solved so far have been small, well-defined, and performance was not an issue. The TAMPR system has been successful for optimizing sequential programs written in FORTRAN. GIST system transforms database-like specifications into data structures and code.

Other transformation systems discussed in the literature have also been successful in limited ways. The greatest success has been in areas where mathematically sound ideas have been made practical. Parser generators and syntax-directed editor generators have been implemented using attributed grammars. Systems that transform code written in one programming language into another programming language have also been successfully implemented.

For well-known, specialized application domains, it is possible to build (over time) a system to transform a specification written in a special-purpose specification language into executable code. When the application domain becomes more general or where experience concerning the domain is lacking, there has been no record of successful implementations of transformation systems. Any recorded success, as in DRACO, has been on very small problems.

The RaPIER project has the option of pursuing research in the area of fragment generation or waiting for the technology to mature. The first choice means that we improve upon what is available now and adapt it to our work.

In order to adapt anything to our work, we would want to examine our motivation for prototyping in the first place. We are building prototypes that will be used by software developers that build software for embedded computer systems. For such systems, synchronization and timing are important. We note that there has been no claim of successful implementation of a transformation system that generates programs with concurrent processes or timing constraints. Even if there were, there is no guarantee that such a system will be adapted quickly and easily enough to mesh into RaPIER's construction methodology.

The TAMPR system is the result of years of experience in building mathematical software in FORTRAN. First, code was stored, maintained and reused. The difficulties of this approach led to the improvements that brought TAMPR about. The RaPIER project is starting out in a more advanced fashion; there is a very high-level language for composing reusable components, programs so constructed can (at least in part) be modified by modifying the VHLL. There has been no experience, however, in the use of this VHLL or the Software Base. There is consequently no documentation for the kinds of problems we run into in the use of the Software Base.

This section suggests that RaPIER recognize that leverage comes with generating code from high-level specifications rather than reusing code. But in order to manage the scope of the present project, it is advisable to build a system in which reusable code can be successfully used for prototyping embedded computer systems software. The next step is to make sure, by domain analysis, that the Software Base does indeed contain the right software for embedded computer systems. When the right set of software becomes available, we can then experiment with abstractions of these in order to define an appropriate VHLL for ECSs and the transformation of that VHLL's constructs into code.

7.5 FUTURE WORK

It is clear that RaPIER must invest in specification languages. PSDL, the Behavior Abstraction classification scheme, and the need for fragment generation all demand that kind of investment. Work on specification languages should start in parallel with the the current RaPIER project; an initial exploratory research should then set the stage for future thrust in this important new area. Since this exploratory research has not yet been conducted, we do not have a program plan for the research. Beacause RaPIER should eventually have the capability of generating code from specifications, we will

- o track the most mature program transformation systems such as GIST
- o bring one of the systems "in-house" for investigation
- o adapt that system to the RaPIER environment if the investigation shows that the system has promise for our work.

blank back page

SECTION 8

DEMONSTRATION/RESEARCH EXAMPLES

8.1 PROBLEM STATEMENT

RaPIER's goals are to develop a prototype engineering environment and to ensure transfer of this engineering environment from sheltered research surroundings to the production milieu (see subsection 1.2).

Meeting these goals is challenging in two respects:

- o requirements of engineering environments for building prototypes are not well understood (which is implied by the fact that none have been built), and
- o technology transfer is often unsuccessful [IEEE83].

Our approach to achieving these goals is to use examples from actual Honeywell projects to guide RaPIER methodology and tool development and to use the results of working on these examples to influence further RaPIER development (see subsection 1.3). The examples we choose serve to:

1. direct and constrain our research and development so that we tackle problems whose solutions meet the real needs of Honeywell divisions and the DoD contractor community in general;
2. test our solutions to those problems, thus assuring that the solutions do work;
3. facilitate technology transfer by involving the natural recipients of this technology in its development, especially in determining its requirements;
4. enable us to demonstrate progress with more than written reports.

8.2 OUTCOME

During this past year we developed three prototypes: two of facets of Honeywell's Space Station work, and one of a fragment of the RaPIER environment. The RaPIER environment prototype is described in section 9). We worked

on the Symbolics 3460(1) using their flavor system as our software base of reusable components and their development tools. The examples did help us focus our attention on real problems and allowed us to test our solutions to them.

Each of the areas in which we are working benefited from the examples. By attempting to use the prototyping methodology and watching how we actually built and used the prototypes, we were able to test and modify the construction methodology. We generated PSDL requirements in part by looking at the information we needed to specify for the ASCLSS and IDA prototypes. Watching how we used the Symbolics tools and where they came up short indicated the kind of tool support RaPIER needs. Building prototypes helped characterize objects in the software base and validate some of the reusability requirements (see section 6). In addition, by doing examples, we gained a better understanding of space station and human factors application areas.

The RaPIER project's first example is taken from the Automated Subsystem Control for the Life Support System (ASCLSS) program at Honeywell's Space and Strategic Avionics Division [HONEYWELL84]. The ASCLSS program is part of NASA's space station effort. With the ASCLSS prototype, we began evaluating RaPIER's proposed prototyping methodology and tools, and understanding the Symbolics host environment. We compared the characteristics of Lisp flavors with the Ada reusability guidelines and refined our software base functional requirements.

RaPIER's second example is taken from the Integrated Display Assembly (IDA) project also at Honeywell's Space and Strategic Avionics Division. IDA was originally designed as a crew interface for payload control aboard NASA's space shuttles. IDA is currently being demonstrated as a crew interface for a guidance, navigation and control application aboard NASA's space station. With the IDA prototype, we evaluated the prototyping methodology in more detail than with ASCLSS. In particular, we looked at the construction methodology, the object-oriented approach, the white-box specification style, and the requirements for the very high level system description language (PSDL). We also began collecting notes on how the prototype was used by human factors experts. This information will be used in developing an execution methodology.

8.3 CRITERIA FOR RAPIER DEMONSTRATION/RESEARCH EXAMPLES

There are four categories of criteria for RaPIER research and/or demonstration examples. Examples must meet as many of the criteria in each category as possible. The criteria are:

(1) Trademark

1. Examples must be characteristic of systems that will be prototyped using RaPIER.
 - a. Examples problem must be large enough so that we are not constructing toy prototypes, and difficult enough so that we are challenged by constructing them. They must be research drivers that will help us answer questions about how prototypes are built, the kind of software that is and can be reused, and the methods of categorizing software components.
 - b. The development of examples on RaPIER must be consistent with the RaPIER objectives listed in subsection 9.6.1. The purpose of the examples must be to identify end-user requirements for embedded computer systems.
 - c. Examples must not actively pursue any RaPIER non-objectives (see subsection 9.6.2).
 - d. Examples must not require specialized hardware, although they may model such hardware in software.
2. Examples must be applicable to RaPIER's intended users (see subsection 9.6.3).
 - a. Examples should be taken from Honeywell divisions. It is highly desirable that examples come from a division that is a member of the RaPIER Technical Advisory Panel (RaPTAP).
 - b. Examples must model the types of problems found in RaPIER's setting (see subsection 9.6.4).
3. Examples must be good demonstration vehicles.
 - a. Examples must generalize a class of behaviors into a manageable problem, as the dining philosophers problem does for resource sharing.
 - b. Examples must have a visual representation, or be able to have one imposed on it for demonstration purposes.
 - c. Examples should be intriguing and puzzle-like.
 - d. Examples must be ones in which the problem as well as many of the difficulties in solving it are "immediately" evident.
 - e. Examples must demonstrate that:
 - fragments built according to the reusability guidelines are indeed reusable for prototyping,
 - software fragments can be classified using the classification scheme and that this scheme is useful for retrieving components from the software base,
 - the (behavior of the) overall architecture of the prototype construction system is sufficient,
 - fragments can indeed be glued together using the methodology described in the glue study,
 - a prototype can be constructed according to the construction methodology,
 - fragments not in the software base can be generated according to the fragment construction approach, and
 - the operators in the software base are sufficient for building prototypes from fragments in the software base.

4. Examples must be able to be developed within resource constraints.
 - a. Examples must be small enough to be do-able yet large enough to be interesting.
 - b. Examples must not require a significant amount of domain specific knowledge. A prototype of an example should be easily understandable. However, domain specific knowledge may be needed to build the prototype.
 - c. Much of the code for the examples should be available.
 - d. The fragments used in building an example should be useful for prototyping similar examples.

8.3.1 The ASCLSS Example Meets the Criteria for a Demonstration Example

The ASCLSS example partially meets the criteria for a research/demonstration example presented above. Here are the categories of criteria, and the ASCLSS characteristics that fulfill each criterion:

1. Examples must be characteristic of systems that will be prototyped using RaPIER.

Examples we showed on August 5 1985 is too small to be characteristic of systems that will be prototyped using RaPIER. However, it is part of a larger ASCLSS example that we may build, in consultation with Honeywell divisions which are currently involved in the ASCLSS study project. Even this "toy" has shown us how to organize a prototyping experiment, and how to design a prototype that models the pieces of a system that are needed for requirements investigation. In this case we are investigating whether a functional requirement is complete, so we model the interface at which a crew member performs the function and the system objects that implement the function. The prototype requires no equipment beyond RaPIER itself to implement or demonstrate.

2. Examples must be applicable to RaPIER's intended users.

This example is taken from a NASA requirements study program that was begun in 1983 at Honeywell's Aerospace and Defense Systems and Research Center and is continuing at Honeywell's Space and Strategic Avionics Division (SSAvD). Personnel at SSAvD expressed the opinion that if RaPIER had been available, they would have used it for the 1983-4 study phase.

3. Examples must be a good demonstration vehicle.

The August 5 1985 demonstration is meant to show how RaPIER works, not how to investigate requirements with RaPIER. Therefore, we needed a simple example that would not get in the way of demonstrating RaPIER. Investigating the completeness of a functional requirement provided that simple vehicle. We demonstrated (1) design according to the construction methodology, (2) construction from reusable software parts, and (3) modification by changing the PSDL (Prototype System Description Language) definition of the prototype.

4. Examples must be able to be developed within resource constraints.

Defining the ASCLSS example required some domain specific knowledge that we acquired easily by talking with human-factors specialists who had worked on ASCLSS. Understanding the demonstration requires no domain specific knowledge. 50% of the prototype comprises reusable components from the Symbolics Flavor System, which is our current "software base." The new components (flavors) that we added to the software base (flavor system) are sensor, actuator and database. They are all obvious candidates for reuse in a continuation of the ASCLSS example and in any embedded system. If we continue the ASCLSS demonstration in Ada rather than Lisp, then the Lisp flavors will serve as designs for their Ada implementations.

8.3.2 The IDA Example Meets the Criteria for a Demonstration Example

The IDA example partially meets the criteria for a research/demonstration example presented above. Here are the categories of criteria, and the IDA characteristics that fulfill each criterion:

1. Examples must be characteristic of systems that will be prototyped using RaPIER.

The IDA is a small example, but is large enough to be characteristic of systems that will be prototyped with RaPIER. IDA was used as a vehicle for communicating requirements to developers of the real system. In place of written requirements, a videotape of IDA was used to communicate the requirements to the developers.

This prototype has shown us what kind of information needs to be specified for a prototype and has given us experience organizing prototyping experiments and designing prototypes. The IDA prototype is not characteristic of the entire class of systems we intend to handle in that its purpose is communicating human factors requirements. The IDA prototype requires no specialized hardware. The IDA hardware can be modeled in software.

2. Examples must be applicable to RaPIER's intended users.

This example is part of NASA's space station program at Honeywell's Aerospace and Defense Systems and Research Center and Honeywell's Space and Strategic Avionics Division. The prototype was found to be a useful means of communicating requirements between human factors experts and IDA implementors.

3. Examples must be a good demonstration vehicle.

This criteria is less important for IDA than are the other criteria. We used IDA to understand the prototyping process rather than as a basis for a demonstration to external reviewers. In particular, we used it to look at how we obtained initial requirements, what kinds of modifications

were likely to be made, what information needed to be included in a prototype specification, how well we were able to follow the construction methodology, the prototyping exercising process, and its ability to communicate requirements to developers.

4. Examples must be able to be developed within resource constraints.

Defining the IDA example required some domain specific knowledge that we acquired easily by talking with the human-factors specialists who were developing the requirements. 70% of the prototype comprises reusable components from the Symbolics Flavor System.

8.4 DESCRIPTION OF THE ASCLSS DEMONSTRATION EXAMPLE

The RaPIER project's first example is taken from the Automated Subsystem Control for Life Support System (ASCLSS) program at SSAvD [HONEYWELL84]. The ASCLSS program is part of NASA's space station effort.

This subsection presents a high-level overview of ASCLSS, the expected benefits of prototyping ASCLSS, an overview of the prototyping process, and a description of each of the four major steps in the prototyping process (see section 3): identifying requirements to prototype, constructing a prototype, exercising the prototype, and incorporating results from prototyping into a response to the original requirements.

8.4.1 ASCLSS Overview

The ASCLSS program objectives are to define a generic automation approach for Space Station subsystems and to demonstrate the selected approach in the control and monitoring of the air revitalization group (ARG) of the regenerative environmental control and life support system (ECLSS). ARG comprises three ECLSS processes: a carbon dioxide concentrator, a carbon dioxide reducer, and an oxygen generator.

The selected ASCLSS automation approach is a hierarchy of distributed controllers. System level, process control, and real-time operating system software will be integrated with controller hardware to demonstrate the automated control and monitoring of three ECLSS processes. A crew interface will be included in the automation system to develop and demonstrate how control authority is allocated among the crew, the upper level system controller and the ARG process controllers.

The crew will use the system to:

- o monitor the status of all processes and hardware,
- o change modes,
- o examine sensors/setpoints,
- o examine/override actuators,

- o examine sensor trend data (7 days),
- o respond to alarm/error conditions,
- o review alarms/errors, and
- o examine schematic representations of subsystems.

8.4.2 Question Under Investigation

Developing a generic automation approach requires understanding the tasks that the system will do and be used to support. Tasks on the Space Station are described through scenarios represented by flowcharts. For the August 5, 1985 demonstration, we have chosen to investigate the completeness of the functional requirements for one task: non-critical unscheduled maintenance.

Our hypothesis is that the scenario for handling non-critical unscheduled maintenance (Figure 8-1) is sufficient, and that the generic automation system provides all the functions needed to accomplish this task.

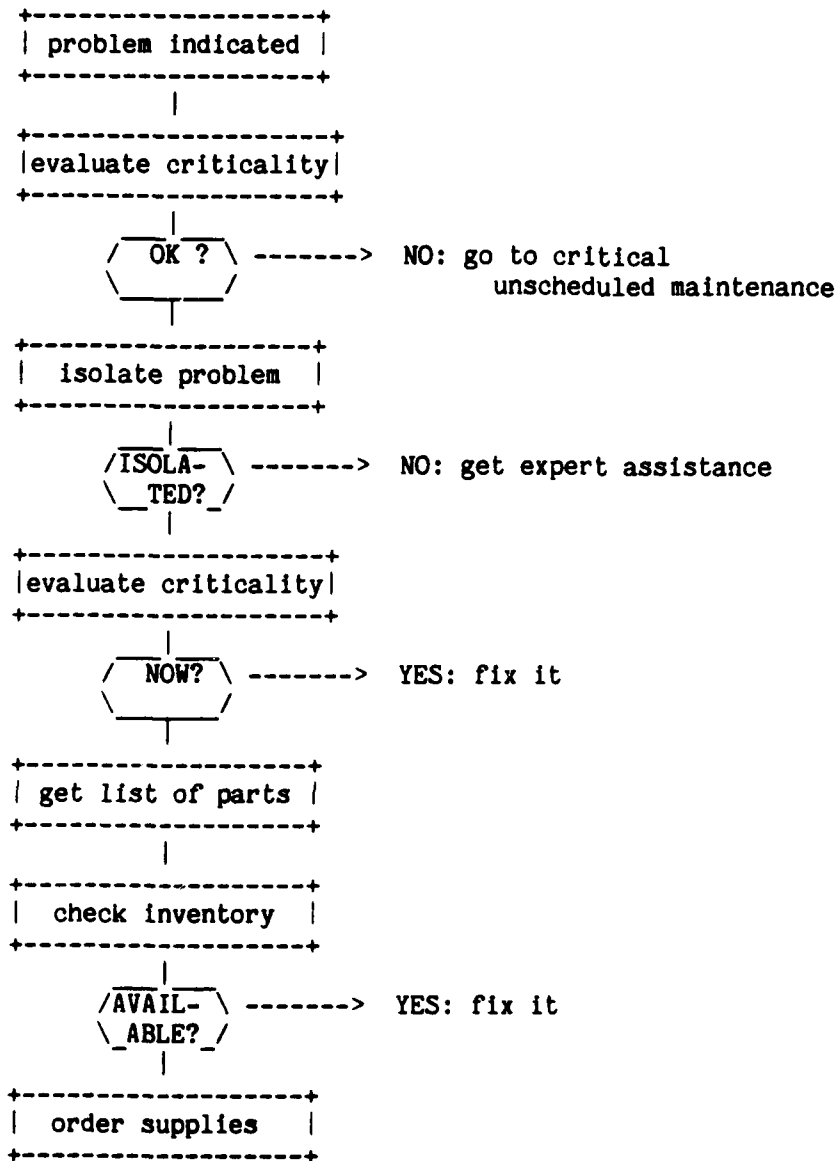


Figure 8-1: Non-critical Unscheduled Maintenance Scenario

8.4.3 The Prototyping Process for ASCLSS

Prototyping is a type of experimentation. Building and exercising a prototype is similar to designing and executing an experiment. These are steps for designing and executing an experiment:

- o Form a hypothesis about the system under study.
- o Create a model of the system that allows you to investigate the hypothesis.
- o Define some metrics or measures on this model.

At this point the experiment is designed.

- o Build the model and execute the experiment.
- o Collect results during experiment execution.
- o Interpret the results and accept or reject the hypothesis.

For the RaPIER demonstration, the system under study is part of the ASCLSS project. The hypothesis is in the form of a question: "Does the scenario for non-critical unscheduled maintenance accurately reflect and support the task to be done?" This hypothesis will be tested by creating a model of the system under study that allows users to walk through the proposed scenario. The model will comprise a display containing three windows, a parts database, a failure log, a prototype environment window containing an alarm and sensors, and a prototype behavior pane. Measures of the model will be English language remarks collected during prototype execution. These results will be interpreted and the hypothesis accepted or rejected. If the hypothesis is rejected, the results will also be used to modify the flowchart or the model of the system. The experiment will continue until the hypothesis is accepted.

8.4.3.1 Identifying Requirements to Prototype

The subset of the ASCLSS system we prototyped was determined by the non-critical unscheduled maintenance scenario. Any part of the system needed to support the scenario became part of the prototype. Because ASCLSS was our first example and is small, we did not put a lot of emphasis on this phase of the prototyping process.

8.4.3.2 Constructing The Prototype

This subsection presents the design and implementation of a prototype to model the part of the ASCLSS system that is needed to test the scenario presented in subsection 8.4.2. The model will allow users to carry out the scenario and will also be modifiable, so that functionality may be added, deleted, or changed in response to users' remarks on their initial interactions with it.

The objects in the ASCLSS model are a subset of the objects in the ASCLSS system. The objects are determined primarily by the scenario under investigation. The model of the ASCLSS system (Figure 8-2) comprises a display containing three windows, a parts database, a failure log, a prototype

environment window containing an alarm and sensors, and a prototype behavior pane. The display is presented as three horizontal windows. The top window is a fixed status display area, the middle window is a call-up display area, and the bottom window is a command area [HONEYWELL85].

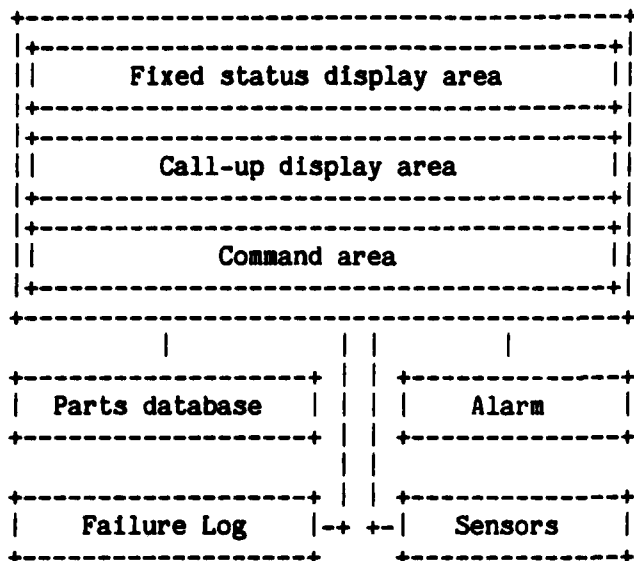


Figure 8-2: ASCLSS Prototype Structure

The ASCLSS prototype was designed according to the operational, object-oriented prototype construction methodology proposed in section 3. The overall design is a "white-box" containing problem-oriented objects. Each object has a set of behaviors that implement the behavior of some real world object. The methodology supposes that the problem-oriented structure of the design will carry over into the prototype implementation. This happened very naturally in the ASCLSS case; the prototype was implemented using exactly the same objects proposed in the design.

We built the ASCLSS prototype in Lisp using the Symbolics's Flavor System as a software base. Lisp flavors are objects in the sense described in section 3; programming with flavors is the Lisp realization of object-oriented program construction [RENTSCH82]. The Flavor System constitutes both software base and software base management system for the approximately 2000 flavors supplied with the Symbolics machine and for all the user-defined flavors.

The Flavor Examiner is the tool for querying and browsing through the Flavor System. It imposes a classification scheme on the collection of flavors in the Flavor System. This is not an explicit classification scheme, as described in [ONUEGBE85b], but a scheme implied by the way flavors are constructed. New flavors (objects) are created by combining existing flavors according to inheritance rules. Flavors are classified in terms of what other flavors they comprise and what methods (operations) they inherit. The Flavor

Examiner (qua Software Base Browser) allows a user to navigate an acyclic graph of the entire Flavor System, where the nodes are flavors and the arcs show relations among them.

The Flavor System's implicit classification scheme is different from the strictly hierarchical scheme now envisioned for the RaPIER Software Base. However, using it is giving the RaPIER project invaluable experience in using a Software Base and Browser long before it can implement its own Software Base Management System. The Flavor System's classification scheme is based on the structure of Lisp flavors (objects); RaPIER's eventual classification scheme will be based on the structure of Ada software.

8.4.3.3 Exercising the Prototype

This section describes the process of working with RaPIER to exercise the ASCLSS prototype in order to investigate the functional completeness of one ASCLSS requirement. This process is presented in the form of a script for the August 5, 1985 RaPIER demonstration. Each item below describes one step in the process; the text in angle brackets describes what happens on RaPIER.

1. <RaPIER shows a welcome message.> The demonstration begins with prototype building. The prototype is a partial model of the ASCLSS system. We use it to investigate the functional completeness of the requirement for handling non-critical unscheduled maintenance.

The objects in the ASCLSS prototype are a subset of the objects in the ASCLSS system. The objects are determined by the scenario we are investigating. The model of the ASCLSS system contains: a display containing three windows, a parts database, a failure log, a prototype environment window containing an alarm and sensors, and a prototype behavior pane.

2. <Select the software base context; select software base window.> The first step in building this prototype is to find objects in the software base that either implement, or can be used to implement, the objects needed for the ASCLSS prototype. We hope to find a window frame, a text display pane window, a menu selection pane, a database, a sensor, and an actuator. The window frame will be used for the display, it will contain three panes: a fixed status display area, a call-up display area and a command area. The first two panes will be instances of text display pane, the third will be an instance of a menu selection pane. Both the parts database and the failure log will be instances of databases. The collection of sensors will be instances of sensors. The alarm will be an instance of an actuator.
3. <Enter "tv:essential-window."> We begin by browsing through the Flavor System (our Software Base) until we find an object, tv:essential-window, whose documentation indicates that its behavior must be part of all ASCLSS windows.

4. <Navigate the Software Base starting from "tv:essential-window."> We now navigate the Flavor System (Software Base) using the Flavor Examiner as a Browser, inspecting flavors (objects) that contain tv:essential-window's behavior, in hopes of finding some to reuse in the display portion of the ASCLSS model. Figure 8-3 shows the paths explored, the flavors selected, and the ASCLSS objects which they implement.

We find three reusable objects:

tv:window pane becomes (ASCLSS) call-up-display-area and
 (ASCLSS) fixed-status-display-area
 tv:command-menu-pane becomes (ASCLSS) command-area
 tv:bordered-constraint-frame-with-shared-io-buffer becomes
 (ASCLSS) display-frame

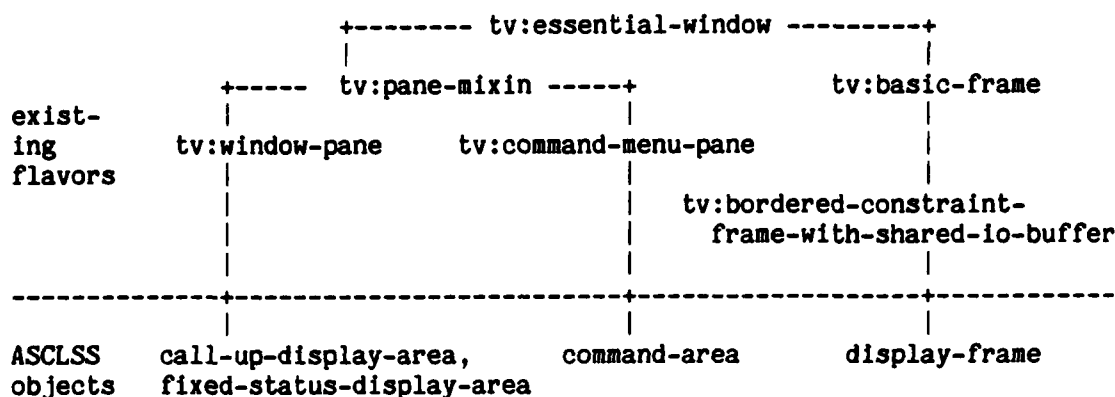


Figure 8-3: Software Base Flavors and the ASCLSS Objects They Implement

5. <Select PSDL window.> We now specify the display portion of the ASCLSS prototype in the high level Prototype System Description Language (PSDL).(1) The display portion of the prototype comprises four objects: fixed-status-display-area, call-up-display-area, command-area, and display-frame.
6. <Select construction context.> Submit PSDL for translation.
7. <Instantiate the display-frame.> We now attempt to instantiate a part of the prototype under construction in order to debug it. The instantiation step is equivalent to the type and consistency checking step that a compiler does on source code. We note that the display-frame object

(1) PSDL and its automatic translation into source code have not yet been defined. In this demonstration, we show a possible PSDL formulation of the ASCLSS prototype and translate it into source code manually.

cannot be instantiated. The Symbolics system tells us that display-frame cannot be instantiated unless it responds to the :any-tyi message. Now we must find an object in the software base that will give the behavior of responding to :any-tyi.

8. <Select software base context; select the software base window. Find all objects that respond to the :any-tyi message.> We will change the prototype's PSDL specification by looking for all objects that respond to the :any-tyi message and choosing one, based on its specification. This matches the classification proposal in [ONUEGBE85b], which says that objects will be classified by the messages they understand.
9. <Select the construction context.> Modify the PSDL of the ASCLSS prototype. Submit the modified PSDL for translation. Instantiate the display-frame and determine that it works properly.
10. <Navigate the Software Base looking for other parts of the ASCLSS prototype.> Look for a database flavor. Look at all methods for the database flavor; the methods constitute the flavor's specification as well as part of its executable code. Look for an actuator flavor. Look at all its methods. Look for a sensor flavor. Look at all its methods.
11. <Select the PSDL window.> Specify the portion of the ASCLSS prototype in which the database flavor is used for the failure log and the parts database, the actuator flavor is used for the alarm, and the sensor flavor is used for the ASCLSS sensors.
12. <Select the construction context.> Submit the PSDL for ASCLSS for translation. Instantiate The ASCLSS prototype.
13. <Select the execution context.> Execute the prototype following the scenario for non-critical unscheduled maintenance.
14. <Select the prototype environment window.> Here we change the prototype's environment so that we can observe its behavior in a different environment. Put a sensor into a warning state, and follow the unscheduled maintenance scenario with the environment (i.e., the sensor) in that state. This leads to executing different parts of the ASCLSS prototype.
15. <Select prototype execution window. Select the remark window; type remarks.> As we exercise the prototype with a sensor in a warning state, we note four possible improvements in ASCLSS's user interface and functional behavior.
 - a. Using the failure log, note that it is ordered by time and not sorted by sensor, and remark that a sort capability on the log would be useful.
 - b. When getting a list of parts, note and remark that there must be a space station wide code number for each part.
 - c. Note that there is no way to schedule maintenance. Suggest that the unscheduled maintenance scenario show a "schedule-maintenance" task.

16. <Stop the prototype. View remarks.> This completes the first execution of the prototype. Look at and interpret remarks and make changes to the prototype based on the remarks.
17. <Use software base and construction contexts to create the next version of the ASCLSS prototype.> The next version of the prototype has a sorted failure log, uses a common coding scheme for parts, and can be used to schedule maintenance.
18. <Select the execution context; select the prototype control window.> Repeat prototype execution using the new prototype. Again put a sensor into a warning state. This time note other deficiencies and remark on them.
19. <Stop the prototype. View remarks.> This completes the execution of the second prototype. Look at and interpret remarks. Decide that it is not useful to continue prototyping.

8.4.3.4 Incorporating Results from Prototyping

The results of the prototyping session described in the previous section are captured in the remarks made by the user. These remarks would be reflected as changes to the initial requirements.

8.5 DESCRIPTION OF THE IDA PROTOTYPE

RaPIER's second example is taken from the Integrated Display Assembly (IDA) project also at Honeywell's Space and Strategic Avionics Division. IDA was originally designed as a crew interface for payload control aboard NASA's space shuttles. IDA is currently being demonstrated as a crew interface for a guidance, navigation and control application aboard NASA's space station.

This subsection presents a high-level overview of IDA, the expected benefits from prototyping IDA, an overview of the prototyping process, and a description of each of the four major steps in the prototyping process: identifying requirements to prototype, constructing a prototype, exercising the prototype, and incorporating results from prototyping into a response to the original requirements.

8.5.1 IDA Overview

The objectives of the original IDA program were to build a crew interface for payload control aboard NASA's space shuttles which:

- o reduced shuttle payload development time and costs;
- o reduced shuttle ground turnaround time;
- o allowed greater flexibility to the mission by accomodating late changes;
- o reduced crew training time/investment [BLOCK84].

IDA comprises a thin film electro-luminescent graphics display, 20 membrane keys surrounding the display, 8 relegendable keys, a 12 key numeric keypad, a 12 key keypad with application specific labels, a 5 key cursor keypad, an enter key, and a 3 key priority-key keypad (Figure 8-4).

Currently, IDA is being evaluated and demonstrated as a crew interface for a guidance, navigation and control application aboard NASA's space station. User interface design experts at Honeywell's Systems and Research Center are developing a set of requirements for a guidance, navigation and control application demonstration on IDA.

8.5.2 Question under Investigation

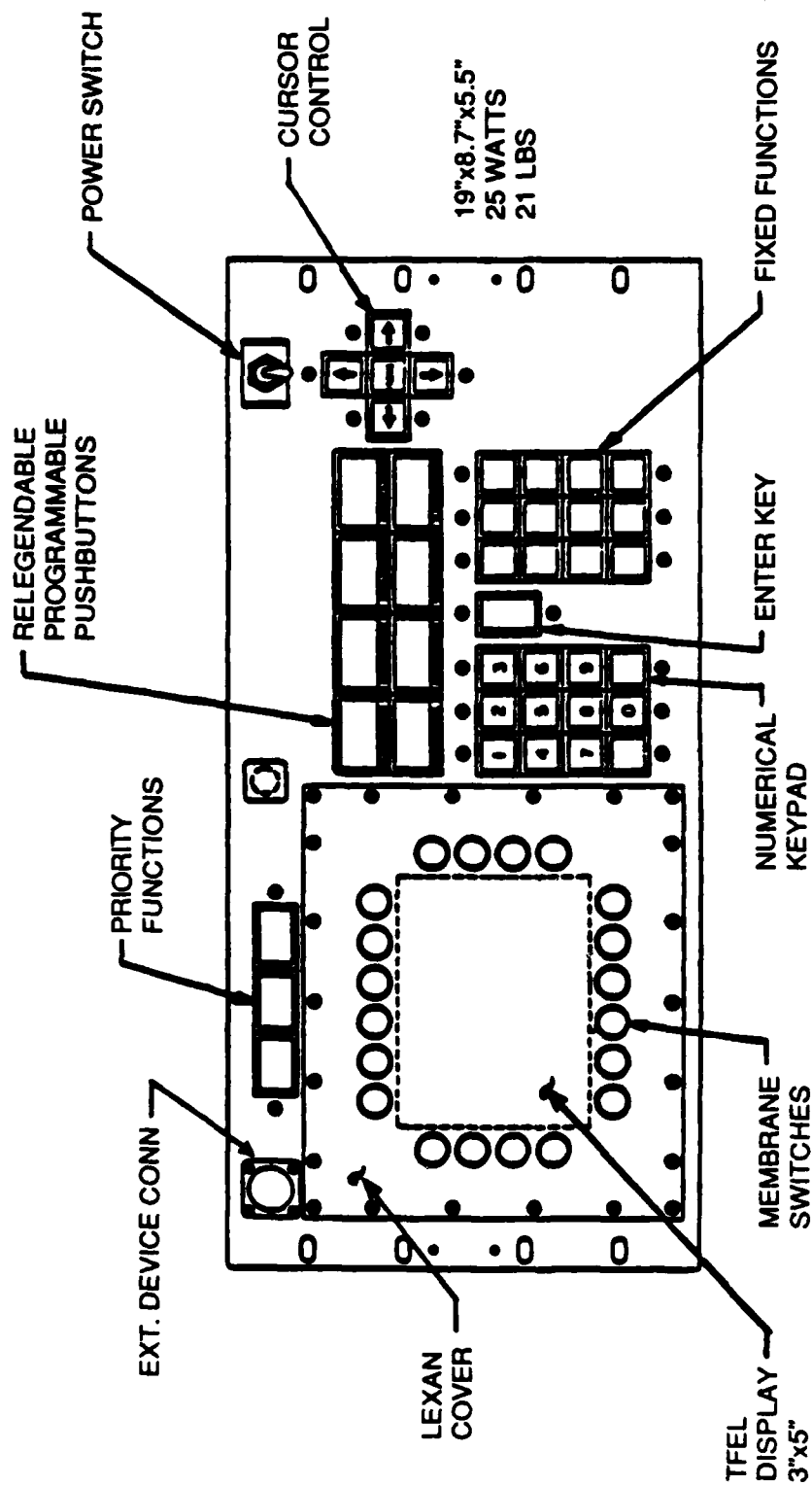
We consider requirements prototyping to be an experiment where some aspect of the requirements is investigated. For the IDA prototype, we have chosen to investigate the completeness and consistency of the requirements specified by the human factors design experts. For this project, we proposed using RaPIER to support building a prototype for communicating the requirements to implementors. It is our contention that:

- o the IDA requirements can be communicated at least as effectively by a prototype as with current methods, and
- o the IDA prototype will uncover conflicting, missing and unrecognized requirements earlier than with current methods.

For the IDA prototype, our hypothesis is that the requirements specified by the human factors design experts are complete and consistent.

8.5.3 The Prototyping Process for IDA

Prototyping is a type of experimentation. Building and exercising a prototype is similar to designing and executing an experiment. For the IDA prototype, the hypothesis is in the form of a question: "Are the requirements specified by human factors design experts complete and consistent?" This hypothesis will be tested by creating a model of the system under study that helps the human factors design experts to visualize the demonstration they are building. Measures of the model will be remarks collected from human factors experts while they are exercising the prototype. These results will be interpreted and the hypothesis accepted or rejected. If the hypothesis is rejected, the results will also be used to modify the the model of the system. The experiment will continue until the hypothesis is accepted.



File No. 6-0367

Figure 8-4: IDA (Integrated Display Assembly)

We followed our proposed prototyping methodology as closely as possible for the IDA prototype. The first step in the prototyping process is obtaining initial requirements. For this step, we talked with the human factors experts who were developing the IDA demonstration.

The second step in the prototyping methodology is constructing the prototype. We built the prototype according to the construction methodology: a white-box, object-oriented approach where the structure of the prototype is the same as the problem-oriented view. In building the prototype, it became obvious that there were many inconsistent and missing requirements. This, however, was expected somewhat due to the need for early versions of the requirements.

The third step in the prototyping methodology is exercising the prototype. We interviewed the human factors expert as he used the prototype.

The fourth step in the prototyping methodology is incorporating the prototyping results into the initial requirements. In this case, the updated requirements are in the form of a videotape of the modified prototype.

The four steps in the prototyping methodology are evident in the IDA prototype. The steps are not sequential. There is a lot of iteration and backtracking. In IDA's construction, when it became obvious that an initial requirement was inconsistent or missing, it was necessary to return to the first step of obtaining initial requirements. We did this by talking to the human factors experts and by making intelligent guesses as to what the requirements might be. Exercising the prototype with a human factors expert included both clarifying and changing the initial requirements. Making these changes meant returning to the second step of constructing the prototype. The decision to make a videotape was a decision to end the prototyping process.

8.5.3.1 Identifying Requirements to Prototype

The requirements investigated by the IDA prototype dealt with the functions assigned to IDA's keys and the positions and contents of fields on IDA's screen. It was a requirement for the IDA prototype to be modifiable along these dimensions. Binding functions to keys and fields to screen positions assume the existence of IDA hardware. The prototype must include the visible IDA hardware (that is, keys and screens).

8.5.3.2 Constructing the Prototype

The first step in constructing the IDA prototype was identifying the problem-oriented objects and their structure. The objects in the IDA prototype design are a graphics display, 20 membrane keys surrounding the display, 8 relegendable keys with 3 large or 12 small characters each, a 12 key numeric keypad, a 12 key keypad with application specific labels, a 5 key cursor keypad, an enter key, and a 3 key priority-key keypad (Figure 8-4). The several kinds of keys are simulated by software. The structure of the

prototype is the same as the structure of the system as viewed by the human factors experts. Because most of the objects are visible, attributes such as size and position can be specified.

Next, the behavior of each of the objects is defined. Since there are several objects of each type, the type of each object is identified. Defining objects behavior in an object-oriented paradigm is equivalent to defining the methods the object will respond to. In the case of IDA, all types of keys respond to the "press" message. Their behavior is determined by a function that causes some effect on the screen, relegendable keys, or audible bell. In addition, relegendable keys respond to the message "set-label" which has the visible effect of changing the label on the keys. The screen responds to many screen-like functions including: "set-cursor-position," "get-cursor-position," "clear-screen" and "output-string." The communication structure is simple. The user sends messages to the keys. The keys affect their environment by sending messages to the screen, relegendable keys, and audible beeper.

After some initial design of the prototype, specifying any more detail about the behavior of the objects became difficult. At this point, the software base was searched for objects that were "close" to the objects needed for the prototype. The Flavor System on the Symbolics served as our software base. Since most of the objects in the prototype are visible, they are based on tv:window-pane. New objects types were created for a generic key type which is based on tv:window-pane, for each type of key (which are based on the generic key type), and for a screen type which is based on tv:window-pane. The new types contain tailoring information and methods that are specific to the new object. Creating new types for screen and key rather than using instances of tv:window-pane has the advantages of making the prototype more easily modifiable and making the prototype implementation structure the same as the problem-oriented structure.

The desired behavior of objects in the prototype is then refined based on the knowledge of existing components and the needs of the prototype. This process of working top-down then bottom-up continues until the desired behavior of the prototype objects converges with the actual behavior of the available components.

8.5.3.3 Exercising the Prototype

When a prototype of the initial requirements was complete, we interviewed a human factors expert while he viewed and manipulated the prototype. The process of exercising the prototype uncovered many inconsistencies. For example, specifications of the screen formats were written on paper about 90 characters wide. Another requirement was that the screen was 53 characters wide. When prototyped, many of the screen formats were clearly unacceptable. As another example, cursor movement was unspecified in the initial requirements. Pressing the move left key 40 times to get from the right of the screen to the left of the screen was clearly unacceptable. This led to a new requirement.

8.5.3.4 Incorporating Results from Prototyping

After many iterations and much backtracking, we ended the prototyping process and made a videotape to record the state of the final prototype. That videotape was delivered to the implementors of IDA. We have yet to evaluate the effectiveness of the videotape and prototype as a vehicle for communicating requirements to the implementors of the IDA demonstration.

8.6 BENEFITS OF EXAMPLES TO THE RAPIER PROJECT

Each of the areas in which we are working benefited from the examples. Through attempting to use the prototyping methodology and watching how we actually built and used the prototypes, we were able to test and modify the construction methodology. We generated PSDL requirements in part by looking at the information we needed to specify for the ASCLSS and IDA prototypes. Watching how we used the Symbolics tools and where they fell short indicated the kind of tool support RaPIER needs. Building prototypes helped characterize objects in the software base and validate some of the reusability requirements (see section 4). In addition, by doing examples, we gained an understanding of space station and human factors application areas.

This subsection describes benefits of examples to the prototyping lifecycle, the RaPIER prototype engineering environment, the software base and reusability guidelines.

8.6.1 Prototyping Lifecycle Benefits

Many of the benefits of building prototypes are methodology related. RaPIER's prototyping methodology has four phases:

1. identifying requirements to prototype,
2. constructing the prototype,
3. executing/exercising the prototype, and
4. incorporating results from the prototype.

The benefits to each of the four phases are described in order.

We found that deciding which requirements to prototype was not always clear to the application area expert. This fact makes a strong case for making flexible and easily modifiable prototypes.

When constructing a prototype, the object-oriented approach was very useful. It allowed the builder to implement the prototype in the same terms as (s)he understands it and localize changes. It was also clear that there was no unique problem-oriented view of the objects in a system. Everybody sees the system from their own perspective. However, people can accommodate to a particular structure that is not exactly like their own concept, if the structure is reasonable. The white-box specification technique turned out to

be very useful in describing a prototype. Both developers and application area experts use structure to describe the prototype. The prototypes also gave us a means of determining information required in a PSDL specification.

While exercising the prototype, it was clear that this step may require more than simply a reaction by the application area expert to the prototype. In the case of IDA, prototype exercising was more like an interview than a reaction.

We did very little work with incorporating results from the prototype into a requirements document. We did however make a video tape of the prototype to use as part of the requirements specification to the implementors of the application on the real IDA hardware.

The benefits of building IDA for the RaPIER project are that we:

- o learned the nature of interface prototyping and the kinds of objects needed for interface prototyping so that we can compare it with other kinds of prototyping,
- o learned about the experiment/modify cycle and building modifiable objects,
- o learned about the execution methodology, including acceptable "response time" for modifications,
- o learned about prototype building from initial customer requirements,
- o observed effects of a prototype vs. a written document as an engineering specification to implementors,
- o learned how human factors experts work.

8.6.2 RaPIER Prototype Engineering Environment Benefits

The benefits to the RaPIER prototype engineering environment are detailed in section 9.4.2. Some prototype engineering environment questions answered by the examples are:

1. What tasks are needed for prototyping?
2. What functions are needed to support each task?
3. How are the functions be partitioned onto windows?
4. What windows are visible simultaneously?
5. Does each window support its task?

8.6.3 Software Base and Reusability Benefits

Many of the objects in the software base are not directly reused. Instead, new types of objects are created to encapsulate parameters needed for a particular prototype. These new objects are very problem specific. Often, an object type exists in the software base, yet a new object type is created. This makes the prototype more easily modifiable and allows the prototype implementation structure to be the same as the problem-oriented structure. Recognition of this sort of object type usage has a significant impact on the kinds of objects we envision to be in the software base.

8.7 FUTURE WORK

We will continue to use examples to drive, evaluate and validate relevant aspects of our work. In the coming year, we will build design prototypes of critical parts of the RaPIER system design and then use an example to validate the RaPIER tools as well as the methodologies and concepts they support.

While searching for appropriate examples, we found that a significant number of Honeywell divisions want (or say they want) design prototypes for evaluating design alternatives. The tools they envision are special purpose tools unique to their work. It is likely that this need will affect RaPIER's future development plans. However, because we feel that requirements prototyping is a prerequisite for design prototyping, we will develop a requirements prototyping capability before tackling the harder problem of design prototyping.

blank back page

SECTION 9

RAPIER PROTOTYPE ENGINEERING ENVIRONMENT

This section presents initial requirements for the RaPIER prototype engineering environment, and an evaluation of those requirements based on our experience in using a fragmentary prototype of RaPIER. The results reported here were achieved under contract task H1.5 and with Honeywell internal funds.

9.1 PROBLEM STATEMENT

RaPIER's goals are to develop a prototype engineering environment and to ensure transfer of this engineering environment from sheltered research surroundings to the production milieu.

Meeting these goals is challenging in two respects:

- o requirements of engineering environments for building prototypes are not well understood (which is implied by the fact that none have been built), and
- o technology transfer is often unsuccessful [IEEE83].

Since we believe that prototyping is the appropriate method for ensuring that a system meets its users' needs, our approach is to develop a prototype engineering environment as a series of prototypes (see section 1.4). We will involve potential RaPIER users in the evaluation of these prototypes in order to help us converge on a subjectively satisfactory environment.

9.2 OUTCOME

Our first major milestone in this area was the completion of the initial RaPIER requirements. We used these requirements as the basis for building a prototype/mockup of the RaPIER prototype engineering environment. We demonstrated this prototype to potential users through on-line demonstrations and a video-tape. We then evaluated this prototype against the initial requirements and against our own reactions to it.

9.3 RAPIER INITIAL REQUIREMENTS

RaPIER is a methodology and automated system for Rapid Prototyping to Investigate End-user Requirements. The RaPIER project is a research and development project: research to investigate prototyping methods and tools; development to implement a prototype of the RaPIER system.

This section reports the first steps in a self-experiment in prototyping to investigate RaPIER's end-user requirements. The experiment comprises:

1. Writing imprecise initial requirements for RaPIER in English. The requirements are imprecise because RaPIER is currently a poorly understood system. RaPIER is poorly understood because research is needed to determine what methods it is automating, and how those methods should be automated.
2. Building a prototype for RaPIER from these imprecise requirements. During prototype building we will
 - o observe how we (the RaPIER team) interpret requirements,
 - o observe what requirements we choose to investigate with the prototype,
 - o observe how we build the prototype, and
 - o measure how many resources (time, money) prototype construction uses.
3. Experimenting with the prototype, and modifying it until we feel (intuitively) it models a subjectively satisfactory prototype-development-system. During our experiments we will
 - o observe what reactions we, and anyone we can convince to experiment with our prototype, have to each version of it,
 - o observe what modifications we make to the prototype,
 - o observe when and how we decide to stop prototyping, and
 - o measure how many resources (time, money) the prototype exercise-modify cycle uses.
4. Incorporating prototyping results (for example, the prototype's code or remarks on it) into an engineering response to the initial requirements - a more complete written document which is the basis for RaPIER's design and implementation.
5. Recording all the observations and measures we take.

9.3.1 Introduction

RaPIER is a methodology and automated system for Rapid Prototyping to Investigate End-user Requirements. The RaPIER project is a research and development project: research to investigate prototyping methods and tools; development to implement a prototype of the RaPIER system. This subsection presents the requirements for RaPIER's automated facilities. The requirements reflect the fact that research is an integral part of the project, and that RaPIER comprises both methodology and automated facilities.

RaPIER's organizational paradigm is the context. A context provides information or services. The user model of contexts is that contexts are like papers on a desk: a context is opened for work and abandoned when a new task is started. The users' model of RaPIER is of a collection, not a hierarchy, of available contexts. Users' interact with RaPIER by selecting the context that provides the desired services, using those services, and selecting another context for the next task. Contexts are manifested through windows on a workstation screen. Basic services may be provided in many contexts by support functions that are available without context switching. For example, the time of day may be available in any context through a function key.

This report presents RaPIER's requirements as a collection of contexts and support functions. Each context represents a desired (set of) capability(ies), not a module in the system's architecture. The device of presenting major requirements as contexts reflects the user-model of RaPIER we wish to promote; the particular contexts chosen (see subsection 9.3.5) are not intended to suggest a system architecture.

There are core and non-core requirements. Core requirements must be implemented in order to have a functioning system. Non-core requirements are either desirable but not necessary functional capabilities, or capabilities that are necessary but infeasible to develop within RaPIER's timeframe, staffing, and funding. Non-core requirements are marked with an exclamation point (!).

Requirements we are unsure of are written in curly braces {}; they will be investigated by prototyping.

The subsection is organized as follows:

- o 9.3.2 discusses requirements and development decisions already made.
- o 9.3.3 presents some fundamental principles of the RaPIER research work and general requirements for the automated system that augment and pervade the specific requirements.
- o 9.3.4 lists RaPIER's objectives.
- o 9.3.5 and 9.3.6 present RaPIER's contexts and support functions.
- o 9.3.7 describes non-automatable methods.

- o 9.3.8 lists some open questions.
- o 9.3.9 discusses development workstations.
- o 9.3.10 discusses prototyping as exploratory programming.

9.3.2 Decisions Already Made

Five decisions have been made that constrain the project's research methods and RaPIER's design and implementation. These constraints help the project by providing early focus. The decisions are:

1. RaPIER is for prototyping only. Prototyping is a kind of exploratory programming activity that produces prototype systems for use by a moderate number of people to test ideas about the requirements under investigation, not about the prototype-program itself. RaPIER is not deliberately designed to be a multi-purpose environment for developing prototypes and other software. RaPIER is designed to support the development of "throw-away" prototypes, not incrementally developed systems. It will be a happy accident if RaPIER turns out to be useful for incremental product-development; however, we will not make deliberate design decisions that serve incremental development.
2. RaPIER's front-end resides on a powerful personal workstation that can be networked to heterogeneous hardware/software. Our prototype version of RaPIER will not be implemented for a time-sharing system, or for a PC type personal computer. See subsection 9.6 note E for a discussion of the benefits of workstations, a definition of "powerful," and a justification of the need for a "powerful" workstation.
3. RaPIER is a distributed system. Making RaPIER distributed from the start is a forced choice, a result of our cooperation with an independent company which is implementing RaPIER's software base (SWB) on a Unix platform, probably an Apollo. RaPIER's workstation front end will communicate with the SWB through a user interface and a programmatic interface. The SWB will be available over a local area or long-haul network. Distributing RaPIER is a burden: our research is in prototyping, not in distributed system development, yet we have to tackle distributed system problems. On the bright side, distributed development systems are the wave of the future; RaPIER will not be obsolete before it is deployed.
4. We will develop RaPIER by prototyping because:
 - o We believe in prototyping as a method for identifying end-user requirements and choose to use the method we believe in.
 - o We advocate prototyping especially when requirements are not well understood; RaPIER's requirements are quite fuzzy at present.

- o We already have a baseline RaPIER prototype (see 5 below) which puts us years ahead of developing requirements from scratch.
- o We propose to develop prototype construction and use methodologies. Methodology is best developed through experiments with real problems. RaPIER's development is a valuable self-experiment.

5. The Symbolics 3460(1) is RaPIER's platform and our experimental vehicle.

As a platform, the Symbolics provides a powerful personal workstation (see 2 above) and a software library of some 10,000 Lisp functions and flavors (abstract type definitions) which

- o play the role of reusable software for software-base research and emulation,
- o are available to incorporate into RaPIER,(2) allowing us to experiment with prototype building by reusing software modules.

As an experimental vehicle, the Symbolics is a well thought out, product quality, development environment. We are also creating a development environment. During requirements definition, we can assume that the Symbolics is our level zero prototype and modify it until we have a prototype for RaPIER. During design and implementation, the Symbolics is a laboratory animal we can poke and prod to find out how the requirements we have identified are realized in one high quality development environment.

9.3.3 General Principles and RaPIER-wide Requirements

These general principles guide our research work:

1. This project is a feasibility demonstration of
 - o constructing prototypes from reusable code,
 - o using prototypes to clarify end-user requirements.

This demonstration is complex enough without the added problems of developing a good development environment, so we will exploit the chosen platform as much as possible.

2. Using our prototyping paradigm to identify RaPIER's requirements implies informal initial requirements statements (like this one), quick prototyping of RaPIER's requirements re-using the Symbolics's software as

(1) trademark

(2) N.B. This implies that at least some of RaPIER, and probably all of early RaPIER, will be developed in Lisp.

much as possible, testing the requirements by applying the RaPIER prototype to sample problems, and modification of RaPIER in response to the results of our experiments. We will work in the same manner as we expect RaPIER's users to work.

3. Useful development environments are based on a theory of how development is done. RaPIER should be based on a theory of how prototypes are constructed and exercised for requirements clarification. RaPIER should model a theory of prototype development that corresponds, as much as possible, to users' intuition about how they go about developing and executing prototypes.
4. It's hard to introduce an unplanned feature into a partially developed system, so we will propose "the world" in these requirements, identify the core elements of the methodology and automated tools, develop those, and provide "hooks" for the world. It may be true that some wish-list-item that appeared hard to provide is in fact easy to implement. Experimental systems should be specified with everything we know how to ask for, because such specifications are not implementation contracts, but only working lists.
5. To avoid severe technology transfer problems, and to keep us aware of what we're about, we will develop user documentation early.

These are some overall requirements on the RaPIER system:

1. (!) RaPIER offers reliable file service.
RaPIER's main file server (either a local or remote, large capacity server) should be available with probability
 $P(\text{file-server-up}) \geq P(\text{one-or-more workstations up})$
which can be restated as
 $P(\text{file-server-down}) \leq P(\text{all workstations down})$

(We will not try to meet this requirement in our initial implementation. It is stated so that platform choices for this and later versions of RaPIER can tend toward a highly reliable file server.)
2. Minor changes are visible quickly.
Minor changes to a prototype during construction or exercise should be visible quickly. This implies that minor changes must be displayed without resetting the prototype to an initial state (such as its start state) and running it until it reaches the state from which the change was made. This requirement implies that dynamic linking is needed in both the construction and the execution contexts.
3. RaPIER is a development environment.
RaPIER should provide the desiderata of any good development environment [TEITELMAN84]. Two such are:

RaPIER Prototype Engineering Environment

- o modeless context switching, because we expect developers to use several cooperating functional capabilities to carry out any task they do;
- o uniformity of user interfaces, because we expect developers to be experienced enough to become experts with RaPIER as they use it, inferring new usage modes from familiar patterns.

Other general development-environment requirements will be identified through experiments with RaPIER's prototype, the Symbolics.

4. (!) RaPIER automates prototype documentation.
As much as possible, a prototype under development should be automatically documented; RaPIER should be able to produce some documentation from internal information about the prototype and its components.
5. RaPIER provides "positive-chatter."
RaPIER must provide ongoing "positive-chatter" to inform users of what's going on (what context the user is in, what options are available with the mouse, what the state of the context is, and so forth). RaPIER should not communicate only negative information. This is especially important since some RaPIER services, such as making a network connection, require large enough amounts of computation to make the system appear to "hang."
6. RaPIER collects bugs and gripes.
Provide a bug reporting and a gripe "mailbox." The mailbox can be a file, an electronic mail connection, or any other means which users can complain into conveniently. This will please testers and pilot project users, will collect useful data for developers, and may prevent some midnight phone calls. During RaPIER's pilot project phase, a gripe address is essential. Since a new system does not stabilize for years, a gripe file can collect information and alleviate user frustration even when RaPIER is transferred into divisions.
7. RaPIER resides on an advanced platform.
We expect RaPIER to survive into the 21-st century; we should not design it for 1970's hardware and software technology.
8. RaPIER is a language dependent system.
Make its functional capabilities serve the PSDL (Prototype System Description Language) or Ada wherever reasonable. For example, provide structure editing capability rather than general text editing capability only.

9.3.4 RaPIER System Objectives

The RaPIER System will

1. automate prototyping for the timely identification of end-user requirements;
2. support prototyping of ECSs in Ada;

3. model a theory of prototype construction and execution;
4. support prototype construction and execution in the defined setting (see subsection 9.6 note C;)
5. support prototype construction and execution for the defined users (see subsection 9.6 note D;)
6. be extensible; the design must accomodate enhancements;
7. be transferrable; that is, meet the expressed need, be documented, be robust, be accompanied by training materials, and be supported by consulting services from its builders;
8. make it easy to reuse Ada software;
9. be a tool that may be used along with other system development tools.

9.3.5 Contexts

The following subsections present the major contexts in RaPIER's conceptual design. They do not suggest an architecture for the system design, but do indicate the users' architectural model of the system.

9.3.5.1 RaPIER Top Level

Function:

1. RaPIER Top Level displays a system "title page."
2. RaPIER Top Level tells users how to reach other contexts.
3. At new invocation, RaPIER Top Level may personalize the user's environment.

Operational Requirements: none

Performance Requirements: Title page text must appear within {one minute} after RaPIER is invoked.

Interface:

INPUT	SOURCE	INTENT
init_select_signal	user or other context	Signals the beginning of a new interactive session between RaPIER and a user. Should cause tailoring of the user's environment if the user desires and if personalization information exists. May cause display of banner information and navigation information.

RaPIER Prototype Engineering Environment

select_signal	user or other context	Signals the selection of this context. May cause display of banner information and navigation information.
personalization_info	user or file server	A collection of parameters that define the user's personal environment.
OUTPUT	DESTINATION	INTENT
title_page_info	screen	banner information about RaPIER. {Details to be determined by prototyping.}
navigation_info	screen	tells user how to get to other contexts (services and information).

Assumptions/Preconditions: Weak assumption: A large-capacity file server is available. The user should be allowed to enter RaPIER even when the file server is unavailable.

Exceptional Conditions: The large capacity file server is unavailable. Personalization information may reside on the file server. Unavailability of the file server should be treated as absence of personalization information and the user should be allowed to enter RaPIER. When environment personalization is requested, a message must be given that personalization was impossible because a file was unavailable, and the user may be given the chance to furnish information interactively.

Remarks:

1. RaPIER Top Level is a hallway from which to go elsewhere. It is needed, at least, because you have to be somewhere upon system invocation.
2. Top Level should provide the user with the means to reach other contexts without requiring the user to request them directly.

9.3.5.2 Context Opener

Function:

1. Context Opener opens a context selected by the user or by a function in another context. If the selected context has not been open before, opening comprises instantiation and opening. If the selected context has been opened before, the context is re-opened in the state in which it was most recently abandoned.
2. Context Opener provides these methods for users to select a context:
 - o function key sequences or mouse-sensitive menu,
 - o (optionally) typed input.

Operational Requirements: Any typed input must be editable using a subset of the system's editor commands. Context Opener may be abandoned; in that case the system remains in the context from which Context Opener was invoked.

Performance Requirements:

1. If the menu is used, it must be displayed within {two seconds}.
2. A requested context must be opened within {one minute}. Note that a context may be open but not available to the user for some time because of initialization activities controlled by the opened context.

Interface:

INPUT	SOURCE	INTENT
u_select_signal	user	Signals that the user wants to change contexts in RaPIER. Triggers acquisition of the name of the target context {and possibly other information, to be determined by prototyping}.
c_select_signal	other context	Signals that a context change in RaPIER is needed. Triggers acquisition of the name of the target context {and possibly other information, to be determined by prototyping}.
target_context	user or other context	Name {and information about} a context. Triggers immediate opening of that context.
OUTPUT	DESTINATION	INTENT
navigation_info	screen	Instrument for the user to choose the context to be opened. Triggers acquisition of a context name {and other information}.
error indication	screen {and/or audio}	indication that target context cannot be opened.

Assumptions/Preconditions:

1. Context Opener has a list of all known contexts, system or user supplied; only contexts on the list are candidates for opening.
2. Context opener may be invoked by the user or another context at any time.

Exceptional Conditions:

1. Nonexistent selection by user: User's input does not name a known context. Error handled by attention-getting visual and/or auditory feedback, and an error message on the screen stating the problem. User is then allowed to choose again. If the user makes no selection, the system remains in the context from which Context Opener was invoked.
2. Nonexistent selection by other context: {Return error message to the other context}.

Remarks: The user model of contexts is that contexts are like papers on a desk: a context is opened for work and abandoned when a new task is started. Abandonment is accomplished by opening a new context. Contexts do not have to be explicitly closed; all abandoned contexts remains in the environment waiting to be opened again. This is unlike the Apple_Mac model, which

requires users to close one context before they can open a new one. This model must be supported by system memory management to assure that the user doesn't run out of workspace too quickly.

There must be context management support that maintains the list of contexts, and adds new contexts to the list.

Only a single context is active at any one time, where active means accessible to the user for interaction. {A possible RaPIER implementation is one in which multiple contexts may be visible though not active. These visible but inactive contexts are either unchangeable or changeable. A changeable context has a program running against it whose progress may change what appears on the screen. That program can do anything which does not require user interaction, including input from and output to devices. If/when that programs needs to interact with the user, the context in which it is running becomes unchangeable. An unchangeable context is one whose associated activity is making no progress.}

9.3.5.3 Prototype Construction

Function: To provide prototype development capabilities (see Remarks) including

1. PSDL (Prototype System Description Language) System containing -

- o PSDL Editor - If PSDL is text, this can be general editing capability in the window, but should be a structure editor for the language. If PSDL is graphical, this can be a general graphical editor, but might have some knowledge of PSDL. (!) The Editor should report syntactic/semantic errors in fragments under development.
- o PSDL Language Processor that
 - gathers input tokens into PSDL statements, and PSDL statements into a PSDL fragment,
 - provides immediate execution of PSDL fragments, and error reporting.
 - {The Processor can interact with the SWB (more powerfully than as a terminal for remote logins) to retrieve needed code.}
- o (!) PSDL Debugger to automate fixing PSDL "programs" (If this debugger is at all sophisticated, ISSI has to provide it.)

2. Ada Language System containing -

- Ada Language Processor - for reusable and newly developed prototype fragments. May be compiler, interpreter, or mixed processor. Should allow {the appearance of} incremental execution of program fragments. Should allow dynamic linking and unlinking of program units.
- Ada Debugger - if prototype developers see Ada code.
- Ada Editor - if prototype developers see Ada code.

3. Software Base (SWB) Interface - remote services for user level interaction with the SWB using the SWB's user interface.
4. (!)Fragment Generation Capability - Fragment generation would replace developing needed code through line-by-line Ada coding with a higher level generation capability analagous to business data processing application generators such as Focus or Nomad.
5. {Prototype Development Tools - like the Symbolics Machine's flavor inspector. The nature of the tools will be determined by prototyping}.
6. {Means to incorporate software from sources other than the SWB into the prototype - these will be worked out by research and prototyping.}

Operational Requirements: Any typed input must be editable using a subset of the system editor's commands.

Performance Requirements: {TBD}

Interface: This interface is the collection of the interfaces of the context's pieces, plus

INPUT	SOURCE	INTENT
select	Context Opener	This context is to be opened. {What "opening" means will be determined by prototyping. A possible implementation is that every context interprets Context Opener's signal and does all the work involved in opening itself. The work necessary will be determined by prototyping.}
{kill	Kill Context	This context is to be killed as defined in Kill Context. One possible implementation is that every context interprets Kill Context's signal and does all the work involved in killing itself. The work necessary will be determined by prototyping. A second possible implementation is that Kill Context does all the work.}

Assumptions/Preconditions: Weak assumption: SWB services are available. The user should be allowed to work in the prototype construction environment even when the SWB is unavailable.

Exceptional Conditions: The SWB is unavailable. The user should be informed of this and allowed to continue.

Remarks: Prototypes are constructed from reusable Ada software parts, stored in a Software Base (SWB) and managed by a Software Base Management System. A prototype developer interacts with the SWB to search for appropriate parts, then specifies the prototype in PSDL. PSDL statements comprise "software base

operators" and the names of reusable parts. The PSDL "program" is translated into an Ada program containing (1) the reusable Ada parts, (2) other Ada statements that glue the parts together into a single executable program. The "other" Ada statements are translations of the software base operators used in the PSDL program. Developers may also acquire reusable components from sources other than the Software Base. (see [FRANKOWSKI85] for a more complete explanation.)

Between prototype execution sessions, the following off-line changes to the prototype will be made in the construction context:

- o changes to the PSDL specification of the prototype. {Eventually the execution context might support changes to the PSDL without an explicit context switch to the construction environment; initially such a context switch will be required. Such support in the execution environment will come from offering construction functions as support functions, similar to editing functions.}
- o changes to the Ada code of a prototype module. {Initially done with an explicit context switch; eventually handled in the execution context without a user requested switch. Such support in the execution environment will come from offering construction functions as support functions, similar to editing functions.} Eventually prototypes should be viewed as PSDL objects, not Ada objects, and the Ada code will be examined and/or changed as often as assembly code is examined when programming in a higher-level language.

We estimate that developing a fragment generation capability is as big a job as the remainder of the RaPIER project.

9.3.5.4 Prototype Execution

Function: To provide prototype execution capabilities including:

1. Incremental execution - used during prototype construction to execute fragments provided to this context by the Construction Context.
2. Prototype execution - the facility constructs a "world" including prototype interaction, comment/reaction and execution control windows. Execution control includes
 - o a checkpointing service that may be invoked by the user,
 - o the ability to modify prototypes' data tables, (NOTE: This assumes table driven prototypes.) and,
 - o the ability to restart the prototype from a checkpoint with new or modified components or data tables linked into the running prototype.

Operational Requirements: {TBD}

Performance Requirements: {TBD}

Interface: This interface is the collection of the interfaces of the context's pieces, plus

INPUT	SOURCE	INTENT
select	Context Opener	This context is to be opened. {What "opening" means will be determined by prototyping. A possible implementation is that every context interprets Context Opener's signal and does all the work involved in opening itself. The work necessary will be determined by prototyping.}
{kill	Kill Context	This context is to be killed as defined in Kill Context. One possible implementation is that every context interprets Kill Context's signal and does all the work involved in killing itself. The work necessary will be determined by prototyping. A second possible implementation is that Kill Context does all the work.}

Assumptions/Preconditions: {TBD}

Exceptional Conditions: {TBD}

Remarks: We will not work on the Execution Context during the first year of the project. We will do enough work provide the execution facilities needed in the Construction Context. That work will be determined by prototyping.

9.3.5.5 Software Base (SWB)

Function: Provides services (see [FRANKOWSKI85] for details) for managing a collection of reusable software including:

1. a query language used to access descriptions of reusable (Ada) parts and the parts themselves,
2. a browser interface used to investigate the contents of the SWB when there is too little information to formulate a precise query,
3. a set of SWB operators that enable developers to compose new programs out of existing modules in the software base,
4. a catalogue containing schema that describe all the modules in the SWB, maintain each module's history, and keep configuration control information,
5. a network interface that provides SWB user-services to a front-end prototype development station,
6. a programmatic interface that allows other programs to query the SWB without user interaction.

Operational Requirements: Must have the same availability as the main file server (the server is described in 9.3.3).

Performance Requirements: Response time to queries {TBD}. Response time for remote procedure calls {TBD}. (Note the assumption that the programmatic interface to the SWB will handle remote procedure call rather than asynchronous communication by message passing.)

Interface:

User controlled: Using the browsing and query facilities through the SWB-supplied user interface which is provided on the RaPIER front end.

Software Controlled: Using the SWB programmatic interface to make remote procedure calls to the SWB management software.

Assumptions/Preconditions: The SWB resides on a Unix machine; there is a network connection between the software-base and the workstation front end.

Exceptional Conditions: Network down; SWB unavailable. See [FRANKOWSKI85] for error messages and conditions returned by the programmatic interface.

Remarks: The extent and quality of the implementation of these requirements depends on how much funding we have to devote to distributed problems and on how much ISSI provides at the SWB end of the communication between front and back ends.

9.3.5.6 User Help and Training

Function:

1. For RaPIER: Provides on-line documentation of all the system's contexts, on-line help that is apropos of the context in which help is requested, and tutorials for some contexts.
2. For a prototype that is executed for requirements investigation: Should provide some documentation and a tutorial about the prototype; may provide some on-line help apropos of place in the prototype.

Operational Requirements: {TBD}

Performance Requirements: {TBD}

Interface

User controlled: {Accessible through some combination of menus, function keys, and typed lines to be determined by prototyping.}

Software Controlled: Any context providing "help" will use the functions provided by this context.

Assumptions/Preconditions: {TBD}

Exceptional Conditions: {TBD}

Remarks: In part, this capability should be a set of lower-level functions that can be invoked by other software to provide help or documentation without necessitating a context switch. We should use the Symbolics 6.0 Document browser as a prototype for the on-line documentation services.

9.3.5.7 (!) Incorporating Results

Function: To incorporate the results of the prototyping exercise into the engineering response to the user's initial requirements by converting the final prototype and information about it into requirements statements. The engineering response is a written document and is the basis for system design and implementation.

Operational Requirements: {TBD}

Performance Requirements: {TBD}

Interface:

Assumptions/Preconditions: {TBD}

Exceptional Conditions: {TBD}

Remarks: This element of the RaPIER methodology will not initially be supported by computer tools. In principle, and with a great deal of research, this step could be automated.

9.3.5.8 Services

The system will provide the "usual" development environment services: mail, general editing, document preparation (WYSIWYG or compose-like). Services to be provided will be determined by prototyping.

Many development services are already available on the Symbolics; for the initial RaPIER system, we will not work on improving what Symbolics offers.

9.3.6 Support Functions

The following sections present support functions which will be available in many contexts.

9.3.6.1 Construction Database

Function: A database for holding and managing the state of the prototype under development. It contains:

- o PSDL (Prototype System Description Language) representation,
- o Ada source/object for pieces of the prototype,
- o prototype documentation
 - user information and training materials,
 - design documentation for the prototype, for example, rejected alternative interpretations of some requirement,
- o user requirements statements, and
- o other information as determined by prototyping.

This is common space shared by developers working on the same prototype. This service does not prevent developers from maintaining a private file space. The service should meet baseline requirements for a development data base [ONUEGBE85a], for example, configuration control.

Operational Requirements: {TBD}

Performance Requirements: {TBD}

Interface:

User controlled: See QUESTION below.

Software Controlled: {Other contexts use DB services by invoking functions in the DB's programmatic interface.}

Assumptions/Preconditions: Weak assumption: A large-capacity file server is available. The user should be allowed to work in RaPIER even when the file server is unavailable. {Limited DB services are available even when the file server where the physical DB resides is unavailable.}

Exceptional Conditions: The large capacity file server is unavailable. The DB may reside on that file server. Unavailability of the file server should not prevent the user from working on construction. A message must be given that the server is unavailable, and limited DB services provided when the file server where the physical DB resides is unavailable.

Remarks: The DB will start out as the Symbolics's file system and some intelligent manual guidelines for its use. It may ultimately be part of the ISSI SWB, or a separate DB connected to or resident on the front end workstation.

QUESTION: Besides being a service, should this be a DB the user can read from directly? write to directly? What concurrency control services are needed in this shared DB?

9.3.6.2 Execution Database

Function: A database for holding and managing the state of the prototype being exercised. It contains:

- o execution checkpoints,
- o remarks about the prototype,
- o versions of the prototype,
- o {other data as needed.}

During prototype execution we expect that the prototyper will not maintain a private file space. Prototype exercise is an experiment. The experiment's lab notebook (the execution database) should hold all the data the experiment generates.

Operational Requirements: {TBD}

Performance Requirements: {TBD}

Interface:

User controlled: See QUESTION below.

Software Controlled: {Other contexts use DB services by invoking functions in the DB's programmatic interface.}

Assumptions/Preconditions: Weak assumption: A large-capacity file server is available. The user should be allowed to work in RaPIER even when the file server is unavailable. {Limited DB services are available even when the file server where the physical DB resides is unavailable.}

Exceptional Conditions: The large capacity file server is unavailable. The DB may reside on that file server. Unavailability of the file server should not prevent the user from limited work on execution. A message must be given that the server is unavailable, and limited DB services provided. See Remarks below.

Remarks: The DB will start out as the Symbolics's file system and manual guidelines for its use. It may ultimately be part of the ISSI SWB, or a separate DB connected to or resident on the front end workstation.

The goal is that prototype exercise be a traditional laboratory experiment during which no private data are recorded outside the "lab notebook," which is the execution database. We wish to record everything that happens during execution, and wish to allow limited execution services even when the DB is unavailable. The nature of the limited services will be decided by prototyping.

QUESTION: Besides being a service, should this be a DB the user can read from directly? write to directly? Will this database be shared concurrently. If it is, what concurrency control services are needed?

9.3.6.3 Save Named Context

Definition: The state of a context is a set of bindings between names and values.

Function: When a user requests the service, save the current state of the context on a permanent storage device under a name by which the user can refer to it.

Operational Requirements: Must have access to a checkpoint preservation server (remote file system, local hard disc, other?)

Performance Requirements: The named context must be saved within {one minute}.

Interface:

INPUT	SOURCE	INTENT
u_select_signal	user	Signals that the user wants to save the context (s)he is currently in. Triggers saving.
{c_select_signal	other context	Signals that a context wants to save itself. Triggers saving. The need for this service will be determined by prototyping.}
context_name	user or other context	a name by which the saved context can be referenced

{NOTE: The execution context may initiate a save for checkpointing during prototype exercise. The Execution Context, or Save Named Context, may provide a name for the checkpoint. Therefore "context_name" may be an input or an output.}

OUTPUT	DESTINATION	INTENT
error indication	screen {and/or audio}	indication that target context cannot be saved.

Assumptions/Preconditions: {TBD}

Exceptional Conditions: Save Named Context may fail if the context preservation server is unavailable. In that case, Save Named Context produces an error message. At the user's request, Save Named Context can keep a request pending and retry it (1) periodically, or (2) when the user requests retry explicitly.

Remarks: This explicit preservation facility allows several users to share a workstation conveniently (one user puts away the other's work, or a user puts away his/her own work), and, most importantly, allows checkpointing during prototype execution.

Saving may be accomplished by active work on this context's part, or by sending a message to the currently active context to save itself. This will be decided by prototyping.

QUESTION: Should Save Named Context be able to save only the context from which it is invoked, or should it be able to save contexts that are referred to in some way?

9.3.6.4 Kill Named Context

Definition: The state of a context is a set of bindings between names and values.

Function: When a user requests the service, adjust the environment so that all objects from the context the user is currently working in are deleted.

Operational Requirements: {TDB}

Performance Requirements: The named context must be killed within {one minute}.

Interface:

INPUT	SOURCE	INTENT
u_select_signal	user	Signals that the user wants to kill the context (s)he is currently in. Triggers killing.
{c_select_signal	other context	Signals that a context wants to kill itself. Triggers killing. The need for this service will be determined by prototyping.}
OUTPUT	DESTINATION	INTENT
error indication	screen {and/or audio}	indication that target context cannot be killed.

Assumptions/Preconditions: {TBD}

Exceptional Conditions: {TBD}

Remarks: Killing may be accomplished by active work on this context's part, or by sending a message to the currently active context to kill itself. This will be decided by prototyping.

9.3.6.5 Restore Named Context

Definition: The state of a context is a set of bindings between names and values.

RaPIER Prototype Engineering Environment

Function: Restore the context named. This context now joins the others in the RaPIER environment.

Operational Requirements: Must have access to a checkpoint preservation server (remote file system, local hard disc, other?)

Performance Requirements: The named context must be instantiated within {one minute}.

Interface:

INPUT	SOURCE	INTENT
u_select_signal	user	Signals that the user wants to open a context which has previously been stored under some name. Triggers acquisition of the name of the target context {and possibly other information, to be determined by prototyping}.
c_select_signal	other context	Signals that a context restoration is needed. Triggers acquisition of the name of the target context {and possibly other information, to be determined by prototyping}.
target_context	user or other context	Name {and information about} a context. Triggers immediate restoration of that context.
OUTPUT	DESTINATION	INTENT
error indication	screen {and/or audio}	indication that target context cannot be restored.

Assumptions/Preconditions: The namespace is partitioned into RaPIER contexts.

Exceptional Conditions: Unknown context name received from user: this facility will report that it has not switched contexts because it cannot find a context with the name given, and provide the user with a means to choose another context or activity. In this case, the user will remain in the mode of requesting context restoration until (s)he chooses to leave it.

Unknown context name received from another context: {this facility will send an error message to the requesting context, and return control to that context. The appropriateness of this behavior will be determined by prototyping.

Restore Named Context may fail if the context exists but the context preservation server is unavailable. "Restore Named Context" will report that. The user or requesting context will remain at the point from which he/she/it invoked Restore Named Context. At a user's request, Restore Named Context can keep a request pending, and retry it (1) periodically, or (2) when the user requests retry explicitly.

Remarks: This facility is not a component of the modeless context switching service mentioned in subsection 9.2.3. Modeless context switching assumes there are many contexts in the environment suspended in the state in which they were abandoned, and allows a user to return to a selected context. This facility restores explicitly saved contexts. This explicit restoration facility allows several users to share a workstation conveniently.

{NOTE: The execution context may put the system in a named construction context for prototype modifications during exercise.}

9.3.6.6 (!) Project Metrics

Function: Measure the cost of prototyping using RaPIER: compute the cost of developing a prototype; compute "imaginary" rework costs assuming RaPIER is not used; compute user satisfaction with delivered system whose requirements were prototyped.

Operational Requirements: {TBD}

Performance Requirements: {TBD}

Interface:

User controlled: In order to establish the true cost of developing and using prototypes, users must report work done on the prototype outside of the RaPIER system. The system developer must also include user satisfaction metrics.
Software Controlled: The software maintains running totals of "on-line" time and other prototype development time. The software helps the user to determine the "what-if" costs of not using RaPIER.

Assumptions/Preconditions: Users report non-RaPIER time accurately. Results are statistical estimates, not absolute figures.

Exceptional Conditions: {TBD}

Remarks: Common wisdom claims that a critical advantage of rapid prototyping is that it lowers costs associated with rework due to poorly defined requirements. These costs may be incurred during development, resulting in increased system costs, or after system release, resulting in poorly performing systems or contract renegotiations. Rapid prototyping also has a cost: the cost of planning, building and using prototypes. Another advantage of prototyping is that systems developed using prototyping are more likely to be well-received by their users [BOEHM84].

While we believe these claims of increased productivity and quality are true, they are not well-substantiated by quantitative data. It is difficult to obtain Honeywell divisional commitment without these data, or to demonstrate ROI for this program to Honeywell or the DoD.

Data gathered using these measures characterize how effective prototyping is in cutting down total system development costs, not how effective RaPIER is among the automated prototyping systems and prototyping techniques available.

The prototype results analysis includes a computation of what costs would have been incurred to develop and enhance the final system if RaPIER had not been used. There are at least two cost models that might be used to estimate these costs: (1) a model which posits using other (ad hoc) prototyping techniques, or (2) a model which supposes prototyping was not used at all.

Function points [ALBRECHT83] could be used as an initial basis for computing costs of not using RaPIER; that is, calculate (new functions + changed functions + f(system size)) to compute costs.

9.3.6.7 (!) RaPIER System Metrics

Function: Measure RaPIER system usage and effectiveness with the parameters described in [CICU83]; for example, tool-usage counts, response-time, or keystrokes.

Operational Requirements: {TBD}

Performance Requirements: {TBD}

Interface:

User controlled: Ideally, RaPIER should include facilities to record and track reasons for changes in the system which was prototyped. This would help us improve RaPIER by determining which requirements were not adequately identified/clarified by using RaPIER.

Software Controlled: The software maintains running totals of "on-line" time, use of each tool, keystrokes, and so forth.

Assumptions/Preconditions: {TBD}

Exceptional Conditions: {TBD}

Remarks: Common wisdom claims that a critical advantage of rapid prototyping is that it lowers costs associated with rework due to poorly defined requirements. These costs may be incurred during development, resulting in increased system costs, or after system release, resulting in poorly performing systems or contract renegotiations. Rapid prototyping also has a cost: the cost of planning, building and using prototypes.

While we believe these claims of increased productivity and quality are true, they are not well-substantiated by quantitative data. It is difficult to obtain Honeywell divisional commitment without these data, or to demonstrate ROI for this program to Honeywell or the DoD.

Data gathered using these measures characterize how cost effective RaPIER is at supporting prototype construction and execution, not how cost effective prototyping is as a development technique. Data gathered using these metrics can be useful in improving RaPIER itself.

9.3.7 Non-automated Methods

The first step of the prototyping process is Requirements Analysis to determine which requirements to prototype. The RaPIER project will propose a methodology for requirements analysis, but will not develop tools to automate the method for lack of time, staff and money. Eventually we may study approaches to automating the requirements analysis method we suggest. One of the approaches may involve an expert system to analyze initial requirements.

User training involves automated services such as on-line help or Computer Aided Instruction Courses, and off-line teaching, tutoring and mentoring. Any development environment is best transferred to new users through a classroom training course. We will develop such a course for RaPIER. Since this is an experimental system that will be applied in pilot projects, we will also be available as tutors and telephone consultants for RaPIER at least through the fifth year of this project. Beyond that, the Computer Sciences Center will need to transfer responsibility for RaPIER, including user training, to a development group within Honeywell.

Documentation includes on-line and off-line material. As much as possible, we will put all RaPIER documentation on line, as Unix(1) does with its "man" pages. That same documentation will be available in hard copy.

9.3.8 Open Questions

These questions will be decided by prototyping RaPIER.

1. Is RaPIER a layer on top of the Symbolics environment, or a tool in the Symbolics environment, co-equal with the other tools? This may be an implementation question that should not be decided in the requirements.
2. Does RaPIER maintain one namespace for all RaPIER contexts (or for all RaPIER contexts plus the rest of the Symbolics software tools) or is the namespace partitioned by RaPIER contexts?

Other QUESTIONS are called out in the definitions of specific contexts which will also be answered by prototyping.

(1) trademark of Bell Labs, Inc.

9.4 DEMONSTRATION EVALUATION

The RaPIER project used the opportunity of presenting a status report to DoD STARS personnel to evaluate the initial RaPIER requirements against the RaPIER prototype built for that status report.

9.4.1 Introduction

The RaPIER demonstration, presented at the August 5, 1985 STARS status report, comprises a prototype/mockup of both the RaPIER system and the ASCLSS example (subsection 8.4). The demonstration is part prototype and part mockup; the prototype part of the demonstration helps us answer questions about RaPIER's requirements; the mockup part exists solely for the demonstration purposes. Figures 9-1 through 9-4 are pictures of the four major RaPIER contexts as they were prototyped in August.

While developing this demonstration, we generated many RaPIER requirements questions. In order to build the demonstration, we had to answer many of these questions. Some of the answers were arbitrary; some were considered carefully. Some were good decisions; others were not. This subsection evaluates these decisions and suggests which should and should not be incorporated into the updated RaPIER requirements. It considers only the prototype part of the demonstration, and focuses on how well that part supports building of prototypes such as ASCLSS.

This subsection has three major parts: an evaluation of the RaPIER demonstration, a comparison of the RaPIER demonstration with the initial RaPIER requirements (see subsection 9.3), and a proposed approach to building the next RaPIER prototype.

9.4.2 Demonstration Evaluation

There are two overall questions we want to answer with RaPIER prototypes:

- A. Is our understanding of the prototyping task complete?
- B. Does the RaPIER prototype engineering environment support the task of prototyping as we understand it?

The next six subsections answer these questions:

- 1. What tasks are needed for prototyping?
- 2. What functions are needed to support each task?
- 3. How are the functions be partitioned onto windows?
- 4. What windows are visible simultaneously?
- 5. Does each window support its task?
- 6. Miscellaneous.

Final Scientific Report: RaPIER Project (Contract No. N00014-85-C-0666)

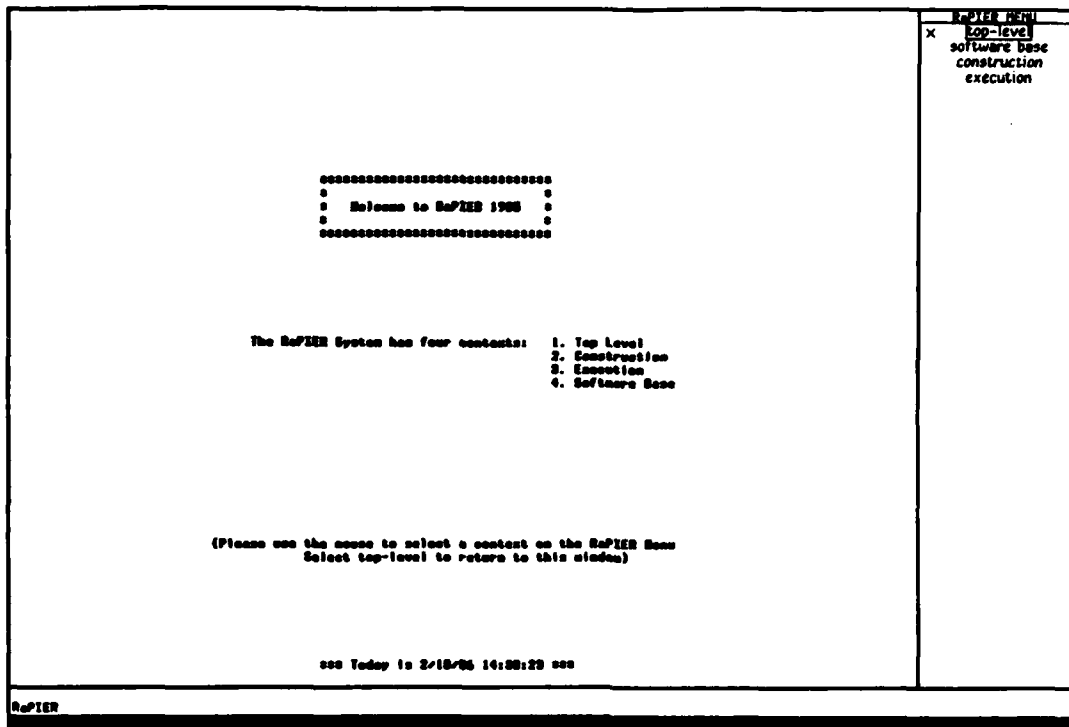


Figure 9-1: RaPIER Top Level Context

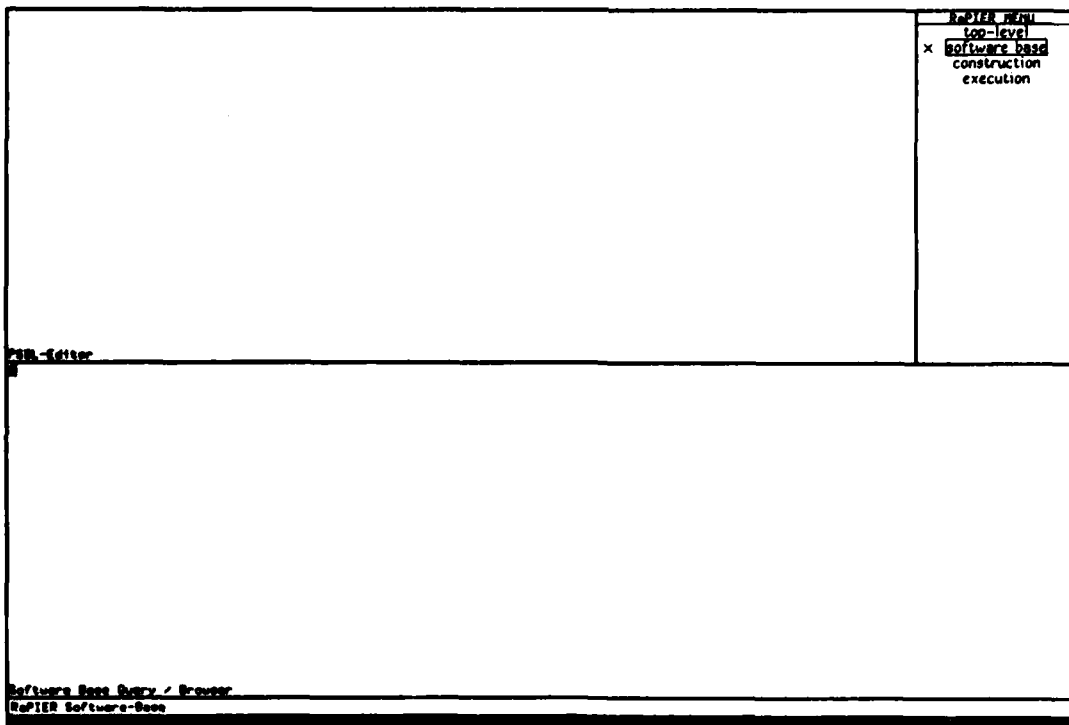


Figure 9-2: RaPIER Software Base Context

RaPIER Prototype Engineering Environment

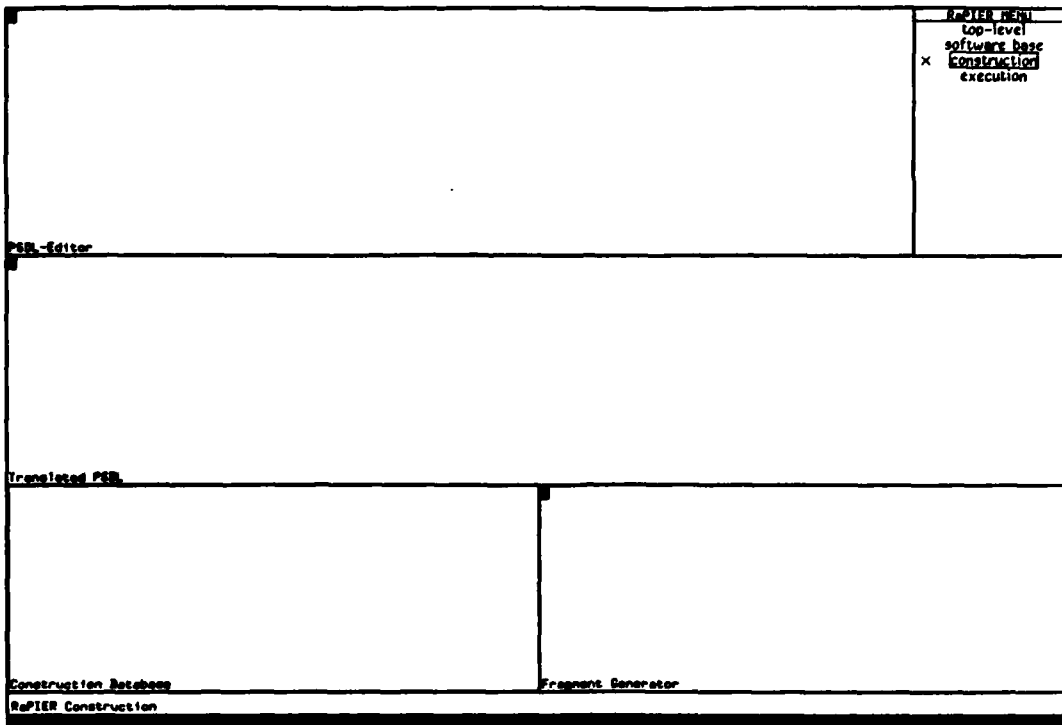


Figure 9-3: RaPIER Construction Context

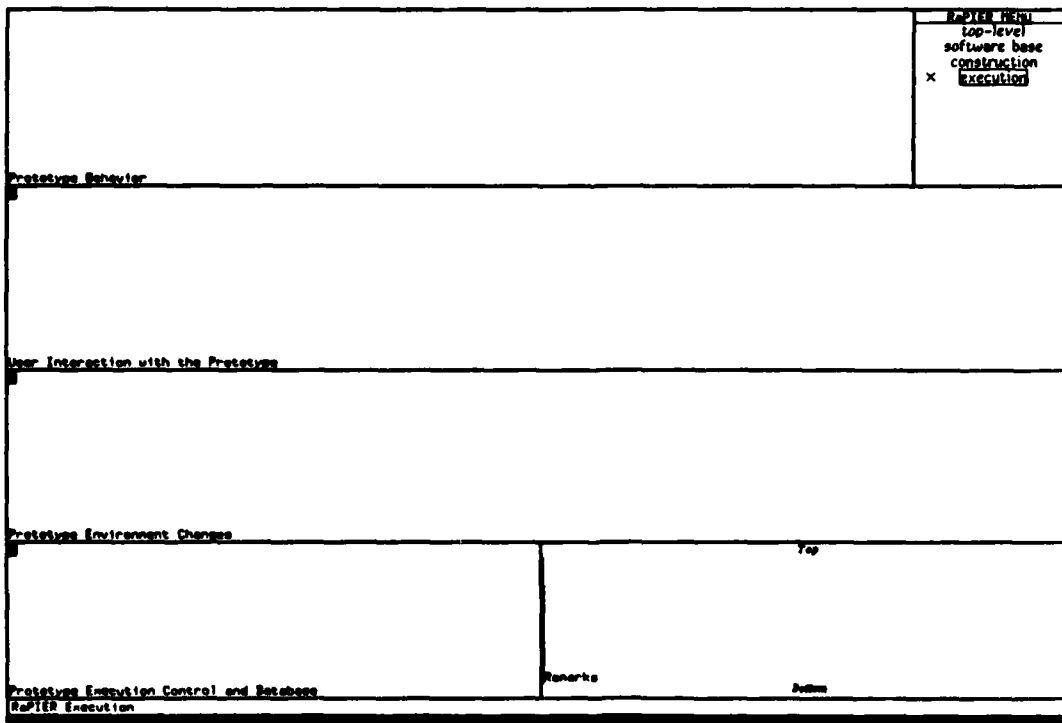


Figure 9-4: RaPIER Execution Context

9.4.2.1 What tasks are needed for prototyping?

In the context of the demonstration, the question is:

- o Does the demonstration script (subsection 8.4.6) describe the task(s) required to build a prototype such as ASCLSS?

The script, which is based on the high level prototyping tasks of the prototyping methodology, describes a particular way to build a specific prototype, but does not show that we understand prototyping in general. The task of building software in general and prototypes in particular is complex and was difficult to capture in a scenario that was intended for a short demonstration.

The script ignores some important parts of the prototyping process. Experience has shown us that, in order to find software components, we need access to experts (possibly ourselves) who have built similar things in similar environments. We hope the need for experts will vanish with a means to query and browse through software component specifications. Other tasks that are not shown in the script are configuration management and analysis of collected results.

Because the script compresses the entire prototyping process into an hour-long demonstration, it distorts the amount of time each task takes. Because the script was intended to lead the audience through the process, it did not mention "side tracks" like configuration management.

9.4.2.2 What functions are needed to support each task?

In the context of the demonstration, the question is:

- o Do the functions in the demonstration support the tasks in prototyping as we understand them today?

The high level functions required to support the task of prototyping are present in the part of RaPIER that we prototyped. Configuration management and a means of gluing components together (i.e., PSDL) are missing. The functions in the demonstration are based on the initial requirements described in subsection 9.3.

A risk is that the functions necessary for a small environment (that is, one with few components, and few ways of doing things) are inadequate for a larger environment.

The need for the capability of modifying translated PSDL is an open question. It SHOULD not be necessary for the developer to know about or modify the prototype's "object" code, that is, an Ada program. However, it may be necessary to have these capabilities, especially during the early stages of RaPIER development.

9.4.2.3 How are functions partitioned onto windows?

In the context of the demonstration, the question is:

- o Are the functions required to do prototyping partitioned into windows in a way that does not require switching windows for "simple" operations?

The functions seem to be partitioned into windows in a way that does not require excessive switching among windows; however, the prototype does not have enough functionality to answer this question well.

There was a lot of switching among windows in the demonstration. This was acceptable since the demonstration compressed the whole prototyping process into an hour. In practice, there may be the same amount of switching among windows, but done over days or weeks. We will find out through experience.

To minimize switching among windows if that is needed, the set of functions available in each window should be extensible.

Until the amount of screen space approximates the amount of available desk space, the amount of switching will be more for an automated system than for paper and pencil and therefore may seem like too much.

9.4.2.4 What windows are visible simultaneously?

In the context of the demonstration, the questions are:

- o Is the organization of contexts as screens with all windows visible appropriate?
- o Is it appropriate to always have the RaPIER context menu visible?
- o Is the set of windows in each context appropriate?
- o Do the set of contexts, and the set of windows in each context minimize switching among contexts?

It appears desirable for all windows in a context to be visible simultaneously; however, constraints known only by the RaPIER user make it necessary to allow the user to choose which windows are visible. Some of these constraints are performance, the need to see information in a window and the need for more windows to accomplish some task. At a minimum, all windows in a context need not be visible; however, they MUST be easily selectable.

It appears appropriate for the RaPIER context menu always to be visible; however, due to limited screen space, another method of selecting contexts may be better.

The set of windows in each context seems appropriate. Some windows may need several instances or multiple buffers. The set of windows in each context should be modifiable by the user.

The windows in a given context are used in conjunction with one another during part of the prototyping process. The organization of windows into contexts minimizes the switching among contexts at the cost of small window sizes.

9.4.2.5 Does each window support its task?

In the context of the demonstration, the questions are:

- o Do the functions provided in each window support the window's function?
- o Does the size and shape of each window support its function?

The prototype is not detailed enough to answer these questions well. It appears that the functions in each window will support the window's function.

In general, the windows are too small and there are too many windows on the screen. The user does not now have control over which windows that are visible and their size.

9.4.2.6 Miscellaneous

In the context of the demonstration, the questions are:

- o Is the use of the mouse and keyboard appropriate?
- o What kind of notification and status windows should be available?
- o What help facilities (on-line help and documentation) are needed?

The prototype is not detailed enough to answer these questions well.

Notification and status windows were not addressed in the demonstration. Notification and status windows are needed in a multi-processes environment. On-line help facilities and documentation are not addressed by the demonstration.

9.4.3 Comparison to Initial Requirements

Here is a brief comparison between the initial requirements and the prototype demonstrated. In general, the demonstration is consistent with the initial requirements.

Requirements	Demonstration
-----	-----
Contexts:	
RaPIER Top Level	Implemented as a window frame
Context Opener	Implemented as RaPIER Menu
Construction	Implemented as a window frame
Execution	Implemented as a window frame

RaPIER Prototype Engineering Environment

Software Base	Implemented as a window frame
User Help and Training	Not part of the demonstration.
Services	Provided by Symbolics

Support functions:

Construction Database	Not part of the demonstration. Implemented as an functionless window.
Execution Database	Not part of the demonstration. Implemented as an functionless window.
Save Named Context	Meaning unclear
Kill Named Context	Meaning unclear
Restore Named Context	Meaning unclear

The meaning of the support functions for saving, killing and restoring a context were unclear and therefore were not implemented.

9.4.3.1 Approach to Building the Next RaPIER Prototype

This subsection suggests approaches to developing the RaIPER requirements. The approach extends our understanding so that we can ask better questions and better answer the questions asked in subsections 9.3.2.1 through 9.3.2.6. The suggested approaches are:

1. Use the Symbolics development tools to develop a prototype (for example, the IDA prototype discussed in section 8) while taking notes on what we do. The notes should include comments about the methodology used to construct the prototype, reusability, software classification and browsing, and gluing components together.

This information will help answer the questions in subsections 9.3.2.1 and 9.3.2.2.

2. Make the functions available in each window easily extensible. Let needs while developing prototypes such as IDA drive how the functions are extended.

This information will help answer the questions in subsections 9.3.2.3 and 9.3.2.6.

3. Do not bind windows to contexts yet. Implement the prototype of the RaPIER prototype engineering environment so that the windows in each context can be changed easily. Some advantages of this are:

- o Each window may have a full screen to work with.
- o Users can configure their environments to their needs.
- o It may minimize the switching among windows because users will choose

- only the windows they will work on for a long time.
- o It will be easier to add or delete windows from contexts while constructing RaPIER.

This information will help answer the questions in subsections 9.3.2.4 and 9.3.2.5.

9.5 FUTURE WORK

We will continue developing the RaPIER environment as a series of prototypes and involving potential RaPIER users in evaluating them. In the coming year, we plan to update the initial requirements with the knowledge gained from our first environment prototype, build a prototype of the updated requirements, and demonstrate this environment to potential users. We also plan to design the RaPIER system and build design prototypes of critical parts of the RaPIER system.

9.6 END NOTES

9.6.1 OBJECTIVES FOR THE RAPIER PROJECT AND THE RAPIER SYSTEM

The RaPIER Project will

1. define a methodology for prototyping to identify end-user requirements;
2. implement a prototype of a system that supports this methodology;
3. transfer this technology, as a prototype, to the defined setting (see subsection 9.6.4).

The RaPIER System will

1. automate prototyping for the timely identification of end-user requirements,
2. support prototyping of ECSs in Ada,
3. model a theory of prototype construction and execution,
4. support prototype construction and execution in the defined setting (see subsection 9.6.4),
5. support prototype construction and execution for the defined users (see subsection 9.6.3),
6. be extensible; the design must accommodate enhancements,

RaPIER Prototype Engineering Environment

7. be transferrable; that is, meet the expressed need, be documented, be robust, be accompanied by training materials, and be supported by consulting services from its builders,
8. make it easy to reuse Ada software,
9. be a tool. It may be used along with other tools, and should be compatible, if not integrable, with those other tools.

9.6.2 Non-Objectives for the RaPIER Project and System

These lists make RaPIER's non-objectives explicit. We will be pleased if we achieve some of our non-objectives as objectives, but we will not make deliberate design decisions in pursuit of non-objectives.

The RaPIER system is not intended for:

1. application generation by programmers or end-users;
2. prototype construction by end-users;
3. developing marketable, or deliverable, products;
4. incrementally building systems that evolve from prototypes into final products;
5. developing functionally complete software systems;
6. creating prototypes with critical resource or timing constraints. Note that a prototype may simulate resource or timing constraints.
7. lowering software development costs. Its objective is to improve quality, thereby lowering total life cycle cost.
8. prototyping systems that will be implemented in an arbitrary language. The system will support Ada concepts.
9. executing prototypes on hardware and operating system platforms that are too small, too "kludgy," or don't offer the services and facilities described in the RaPIER requirements;
10. being used in isolation. We assume the system will be able to reach out to remote software repositories;
11. developing portable (as opposed to reusable) code.

The RaPIER system will not:

1. be based on executable specifications (e.g, PASILey);

2. require a guru to use it.

The RaPIER project is not:

1. developing a Honeywell hardware/software product for sale by Honeywell Information Systems;
2. developing what can be acquired;
3. developing database or reusability technology or tools other than what is needed for prototyping.

9.6.3 The Typical RaPIER User

RaPIER's major components are a construction and an execution environment. This section describes the construction environment's user, unless otherwise noted. The execution environment's user may be less of an expert.

The typical RaPIER construction environment user:

1. is a software developer and computer scientist. Systems engineers and customer subject matter experts may be secondary users.
2. has a B.S. (or good B.A.) in computer science (or the equivalent knowledge);
3. has at least three years of software development experience using a high-level language;
4. has used Ada or will learn Ada prior to using RaPIER;
5. appreciates tools, will learn RaPIER's tools and be motivated to exploit them, can/will become a RaPIER skilled user;
6. deals with requirements;
7. deals with new products;
8. is part of the contractor's staff;
9. is not a casual user. This implies that RaPIER will not support the user who doesn't build familiarity and facility with the system over time. RaPIER will instead require the user to build up facility with the system, exploit on-line help and documentation, and off-line documentation.
10. has skill/experience in designing software, implying that (s)he will plan the prototype and provide an architecture for it instead of just hacking it together. The plan and architecture are what makes reusing software profitable.

11. has some knowledge of the application area, so (s)he can interpret the user's initial requirements in a useful way.

9.6.4 The RaPIER Setting

This list states explicitly the characteristics of the setting in which RaPIER will be used. RaPIER will be used in:

1. a professional, not educational or hobby setting. This implies, for example, that methods must be developed to use RaPIER's results appropriately, mapping information gained from the prototyping exercise into an engineering response to initial requirements, not just concluding the exercise.
2. a production, not research setting. This implies that RaPIER must be reasonably robust, reasonably "useable," and adequately documented. This also implies a certain level of user expectations for functional capabilities such as mail, network links, and text processing.
3. a setting in which the product RaPIER prototypes will most likely be developed in Ada.
4. a large-project setting, where many people work together over long periods of time. This implies the need for the "usual" development environment tools - configuration management, data management, and so forth. The typical Honeywell Training and Control Systems Operation's project, for example, is about 20 person years.
5. a system development, not just software development setting. This may imply that we will have to provide methods for modeling hardware in software, or develop a prototype construction methodology that does not differentiate between hardware and software. This may also imply that some users will be system engineers.
6. a resource starved setting, in which there will never be enough time or money for prototyping. This implies a need for as many prebuilt, reasonable complete prototyping as possible in the software base, and implies that people may want to use the prototype as an initial product. (We hope that, as prototyping begins to pay off, managers and developers may be willing to devote more resources to it.)
7. an embedded system development setting.
8. a for-profit industrial setting. This implies that prototyping must be cost effective with respect to the "risk-cost" of the requirements under investigation. That is, if the potential cost of making an error in some set of requirements is \$200,000 with a 10% likelihood that the error will be made, then the prototype may cost \$20,00 if it drives the risk to zero, \$10,000 if it drives the risk to 5% (half the original risk), and so forth.

9.6.5 Workstations

Here are some benefits of having individual workstations for the members of a prototyping (or development) project:

1. Availability/Reliability:-- Progress is not tied to a single mainframe or mini. If one workstation is down, a developer can use another, getting files from the file server. If the file server is down, a developer can do work in the local file space until the large capacity file server is restored.
2. Availability/long lived sessions:-- A personal workstation is like a desk. It belongs to one person and is an environment that stays the same as long as that person wants it to. Work sessions that generate many files, and many bindings, do not disappear at log-off. Therefore a developer does not have to spend time restoring an environment (with the possibility of mistakes) the next time she/he wants to work on the same problem. This increases developer productivity.

Here are some additional benefits of the Symbolics workstation:

1. Access to colleagues and remote services through a network:-- The Symbolics's normal mode of operation is as a node on a network. This mode provides all the benefits of an individual workstation, plus access to remote services such as mail, and other computers.
2. Power:-- The Symbolics has:
 - o enough memory to support dynamism, and exploratory programming
 - o a variety of I/O devices - at least mouse and keyboard input, screen, file and hard copy output. It has serial interface ports that could be used to provide lightpen or drawing pad input, hard copy graphical output, or input from and output to devices that are peculiar to a system being prototyped.
 - o Reusable components:-- The Symbolics system is composed to hundreds of Lisp flavors. These flavors are reusable packages that can be used to
 - o construct RaPIER,
 - o learn the characteristics of reusable software, and
 - o learn how to build with reusable software.
3. Sophistication:-- The Symbolics system is one of the most advanced software development environments on the market. This project can learn from it, thus improving RaPIER. The project can use large parts of it in the RaPIER environment, thus improving the project's productivity.

9.6.6 Prototyping as Exploratory Programming

Exploratory programming [TEITELMAN84] means trying alternatives. It does not mean hacking; it does not mean debugging. The exploratory programming life cycle recommends designing and implementing in tandem instead of freezing a design and then implementing. The goal of exploratory programming in general

RaPIER Prototype Engineering Environment

is to develop a program to accomplish some task which incorporates the best design decisions of the alternatives tried. Prototyping to identify end user requirements is quintessential exploratory programming: the only goal of prototyping is to test alternatives.

Engineering prototypes (breadboards and brassboards) are nearly complete models of physical objects. They are built to demonstrate concepts and only secondarily to be changed until the right concept is discovered. Software prototypes, on the other hand, are malleable objects that can be changed often, and at low cost if the prototype development system is designed to facilitate change and the prototype is built to be adaptable.

The RaPIER project views software prototyping as the incremental development of a final prototype which may be discarded once system/software requirements are clarified. This kind of prototyping process begins with the incremental development of an initial version of the prototype, done by prototypers exploring their understanding of the user's initial requirements. The final prototype is incrementally developed also, by making changes in response to comments and criticism from application specialists and eventual end users. The whole point of prototyping to identify end-user requirements is to change the prototype often in response to user comments.

blank back page

Financial Summary and Related Efforts

SECTION 10

FINANCIAL SUMMARY AND RELATED EFFORTS

This section describes actual spending, in calendar year 1985, for STARS and for Honeywell parallel efforts in this program. The second part of the section describes some technical efforts within Honeywell which are related and of potential interest to STARS.

10.1 FINANCIAL SUMMARY

STARS funded tasks:

<u>Task</u>	<u>Spending</u>
H1.3 - Reusability	\$35980.75
H1.5 - Demonstration	68946.48
H1.6 - Methodology	23831.63
H1.8 - Planning/Review	23263.11
<u>Total STARS</u>	<u>\$152021.97</u>
Honeywell parallel funding:	\$202868.00
Honeywell cost-share	\$ 77851.94
<u>Total - Honeywell</u>	<u>\$280719.94</u>
<u>Total 1985 Program</u>	<u>\$432741.91</u>

10.2 RELATED HONEYWELL EFFORTS

10.2.1 SOFTWARE DEVELOPMENT ENVIRONMENTS: SDE 1.0

The Software Development Environment (SDE 1.0) is an integrated environment for software development providing support for project planning, visibility into and control of the development process. It supports all members of a software development organization including managers, designers, implementors and maintainers. It provides a common information base which allows all

members of the organization to work together in one consolidated environment. In addition, SDE 1.0 can support Honeywell customers and interactions between divisions.

SDE 1.0 offers managers increased support for software development in the areas of project planning (emphasizing planning a project before starting one), configuration management, monitoring of project activities, product status, metrics collection (i.e., measurement of the productivity of the development effort and quality of its software product, on one project or across multiple projects); a general ability to specify an overall methodology of what will be produced on a project and how it will be produced, and support for transition between lifecycle phases (e.g., design must be done before coding begins).

SDE 1.0 provides designers and implementors with organized information they need to perform project activities (e.g., inputs, outputs, related documentation/activities), the capability to use a wide range of development tools (e.g., communication, text editing, compilation tools), the ability to manage revisions to documentation/source code and maintain change histories, and data to back up the current status of development.

SDE 1.0 provides software maintainers/Honeywell customers with an effective approach to ease the transition into and execution of the maintenance phase of the software life cycle after product development. It can provide a complete context/project history (e.g., every memo, bug fix, piece of source code, version, and so forth that was created during development) to maintenance personnel. It can also help assess the impact of making changes to the software product, and could be used to assess contract performance/quality of delivered software.

Finally, SDE 1.0 is an excellent vehicle for technology transfer because it can be tailored to support development of specific applications. One Honeywell division that has expertise to do software development in a particular area can configure SDE 1.0 to support its specific methods and tools.

In general, SDE 1.0 provides software development organizations with support to do the kinds of things they've always done, but in a more orderly, coherent and consistent manner.

The architecture consists of a data management system, a set of integrated environment functions and an encapsulation interface.

The data management system (Software Lifecycle Information Manager) is based on a schema that is specific to software development. As opposed to traditional development systems which provide users with 'files' and 'directories', SDE 1.0 provides 'projects', 'activities', 'tools', and 'information objects.' Each of these objects has a set of associated attributes and a set of relationships with other objects in the system.

Functions provided by SDE 1.0 fall into six categories:

- o Encapsulation presents a unified functional environment to the user and allows the use of non-SDE tools.
- o Project management is used to set up a project in the SDE environment through the specification of activities; it also provides project scheduling.
- o Object management is used to add, delete, modify, and print information in the database.
- o Configuration management is used to develop software baselines and provide a UNIX- (trademark) like 'make' function. It was not implemented for the prototype SDE 1.0.
- o Revision management provides automatic revision control of textual information objects.
- o Metrics management collects and analyzes data on software development activities, tool usage, and information objects.

Portability was given major consideration during the implementation of SDE 1.0. The current implementation of SDE 1.0 is written in the C programming language using a compiler developed at the University of Waterloo for the Honeywell DPS6 minicomputer. The implementor also observed rigid coding, documentation, and testing standards. Therefore, with minor modifications and an encapsulation hook, SDE 1.0 is portable to any system with a C compiler and a hierarchical file system.

10.2.2 Prototype Reusable Software Repository (RSR)

Work in 1985 to design and develop a prototype Reusable Software Repository served two major purposes. First, it provided insight into a "new" tool for increasing software development productivity. Second, together with the SDE 1.0, it provided us with another step toward our goal of a prototype APSE since a tool supporting reusability of Ada software is a key APSE tool.

RSR requirements were based in part on STARS Application Area reusable Ada software library requirements as well as previous and/or existing Honeywell software repository efforts. These requirements included the ability to store reusable software and information about the software, Honeywell-wide accessibility, and the ability to retrieve, submit, and maintain software inventory items. The RSR conceptual design depicts these functional requirements. It provides a menu-oriented user interface to inventory item retrieval, submission, and maintenance functions. Three queues, a comment/deficiency log, inventory item, and bulletin board separate repository submission and maintenance (insertion) functions to protect the integrity of repository data. Detailed design of the repository was done in an Ada programming design language using object-oriented design techniques.

The prototype implementation of the repository itself uses the INGRES database management system on VAX VMS. It is coded in the C language and makes use of the C interface to INGRES. The implementation supports a subset of overall repository functionality including the ability to browse the repository (categories, inventory items), display inventory item by keyword, display general information about a specific inventory item, view inventory item long description, view actual inventory items, submit inventory item

insert inventory item, delete inventory item, copy inventory item, create category, delete category, and read bulletin board. The implementation is a foundation on which Ada-specific classification schemes for software repositories can be prototyped.

A demonstration of the RSR used sample Ada modules from the SIMTEL Ada repository.

10.2.3 User Interface Prototyper

The objective of this project is to develop and demonstrate a software package which will assist design engineers and human factors specialists in the rapid design and implementation of user interface front panels, in order to reduce the time required to develop and test user interface prototypes and thus the cost of user interface design.

The rapid prototyper was developed in an object-oriented programming language (Smalltalk80) and it contains an inventory of functional and graphical representations of generic man-machine interface (MMI) components which may be linked together to simulate the behavior of a particular device. A graphics editor was developed to facilitate user expansion of the pictorial component inventory. The rapid prototyper's user interface is graphical in nature and incorporates menu-based and form-filling approaches.

In a successful proof of the concept, a cross section of component types was prototyped using Smalltalk-80 on a Tektronix 4404. In parallel with this activity, Honeywell divisions were surveyed to determine the types of MMI components in current or anticipated use and the processes typically used in doing MMI design. The functional aspects of 12 basic and 7 operational components were implemented as well as the capability for the user to define composite components.

AD-A166 353

JOINT PROGRAM ON RAPID PROTOTYPING RAPIER (RAPID
PROTOTYPING TO INVESTIGA. (U) HONEYWELL INC GOLDEN
VALLEY MN COMPUTER SCIENCES CENTER 28 MAR 85

4/4

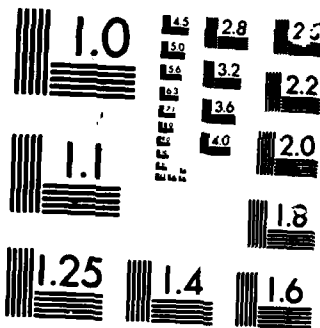
UNCLASSIFIED

N00014-85-C-0666

F/G 9/2

NL





MICROCOPY RESOLUTION TEST CHART

BIBLIOGRAPHY

[ALAVI84]

Maryam Alavi. "An Assessment of the Prototyping Approach to Information Systems Development," Communications of the ACM, Vol. 27, No. 6, June 1984, pp. 556-563.

[ALBRECHT83]

A. Albrecht, J. Gaffney, Jr.. "Software Function, Source Lines of Code and Development Effort Prediction: A Software Science Validation," IEEE Transactions on Software Engineering, Vol. SE-9 No. 6, November 1983.

[ALFORD77]

M. W. Alford. "A Requirements Engineering Methodology for Real-Time Processing Requirements," IEEE Transactions on Software Engineering, Vol.1 SE-3, No. 1, January 1977, pp. 60-69.

[AMANO84]

K. Amano, M. Chiba, A. Mochida and T. Maeda. "An Approach Toward Integrated Algorithm Information Systems," Information Systems, Vol. 9 No. 3/4, 1984.

[BALZER79]

Robert Balzer, Neil Goldman. "Principles of Good Software Specification and Their Implications for Specification Language," Proceedings Specifications of Reliable Software, IEEE Computer Society, 1979, pp. 58-67.

[BALZER82]

Robert Balzer, N. M. Goldman, D. S. Wile. "Operational Specifications as the Basis for Rapid Prototyping," ACM SIGSOFT Software Engineering Notes, Vol. 7, No. 5, December 1982.

[BARSTOW85]

David R. Barstow. "On Convergence Toward a Database of Program Transformations," ACM Transactions on Programming Languages and Systems, Vol. 7, No. 1, January 1985, pp. 1-9.

[BARSTOW85a]

David R. Barstow. "Domain-Specific Automatic Programming," IEEE Transactions on Software Engineering, Vol. SE-11, No. 11, November 1985, pp. 1321-1336.

[BENTLEY85]

Jon Bentley. "Programming Pearls," Communications of the ACM, Vol. 28 No. 7, July 1985, pp. 671-679.

[BERGLAND81]

G. D. Bergland. "A Guided Tour of Program Design Methodologies," IEEE Computer, Vol. 14 No. 10, October 1981, pp. 13-37.

[BIENIAK85]

R. M. Bieniak, L. M. Griffin, L. R. Tripp. "Position Paper: Automated Parts Composition," Proceedings of STARS Reusability Workshop, April 1985.

[BIGGERSTAFF84]

Ted J. Biggerstaff, Alan J. Perlis. "Forward: Special Issue on Software Reusability," IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 474-477.

[BLOCK84]

Roger Block. "IDA: (slide presentation)," Honeywell Space and Strategic Avionics Division, 1984.

[BOEHM83]

Barry W. Boehm. "Seven Basic Principles of Software Engineering," The Journal of Systems and Software, Vol. 3, No. 1, March 1983, pp. 3-24.

[BOEHM84]

B. W. Boehm, T. E. Gray, and T. Seewaldt. "Prototyping Versus Specifying: A Multiproject Experiment," IEEE Transactions on Software Engineering, Vol. SE-10 No. 3, May 1984.

[BOOCH83]

Grady Booch. "Object-oriented Design," Tutorial on Software Design Techniques. Ed. P. Freeman and A. Wasserman, 4th edition (Catalog Number EHO205-5), IEEE Computer Society Press, 1983.

Bibliography

[BOOCH83a]

Grady Booch. Software Engineering with Ada, The Benjamin/Cummings Publishing Company, Inc., 1983.

[BOOCH83b]

Grady Booch. "Dear Ada," Ada Letters, Vol. III, No. 3, ACM SIGAda, November/December 1983, pp. 25-28.

[BOOCH85]

Grady Booch. "ACM SIGAda Tutorial: Ada Methodologies," ACM SIGAda Meeting, July 30, 1985.

[BROWNSTON85]

Lee Brownston, Robert Farrell, Elaine Kant, and Nancy Martin. Programming Expert Systems in OPS5, Addison Wesley Publishing Company Inc., 1985.

[CANNON82]

Howard L. Cannon. "A Non-hierarchical Approach to Object-oriented Programming," M.I.T. Technical Report (Draft), 1982.

[CHENG84]

Thomas T. Cheng, Evan D. Lock, Noah S. Prywes. "Use of Very High Level Languages and Program Generation by Management Professionals," IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 552-563.

[CICU83]

Antonio Cicu. "Proposal of Software Quality and Productivity Measurements: for the SDE Project," Honeywell CSC Technical Report, unnumbered, 1983.

[DOD83]

United States Department of Defense. Reference Manual for the Ada Language: ANSI/MIL-STD-1815A, United States Department of Defense, January 1983.

[FRANKOWSKI85]

Elaine N. Frankowski. "Rapid Prototyping Program Plan, Version II: (in consultation with International Software Systems Inc.)," Honeywell RaPIER Technical Report, unnumbered, Honeywell Computer Sciences Center, Golden Valley, MN 55427, February 1985.

[FRANKOWSKI85b]

Elaine N. Frankowski, Christine M. Anderson. "Design/Integration Panel Report," Proceedings of the STARS Reusability Workshop, April 1985.

[FRANTA77]

William R. Franta. The Process View of Simulation, The Computer Science Library, Elsevier North-Holland Inc., 1977.

[FREEMAN83]

Peter Freeman, Anthony I. Wasserman. "Introduction to Part III: Specification Methods," in Tutorial on Software Design Techniques, IEEE Computer Society, August 1983, pp. 173-176.

[GOGUEN79]

Joseph A. Goguen. "An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications," Proceedings Specifications of Reliable Software, IEEE Computer Society, 1979, pp. 170-189.

[GOGUEN84]

J. A. Goguen. "Parameterized Programming," IEEE Transactions on Software Engineering, Vol. SE-10 No. 5, September 1984, pp. 528-543.

[GOLDBERG83]

Adele Goldberg, D. Robson. SMALLTALK-80: The Language and Its Implementation, Addison-Wesley, Reading, MA, 1983.

[GOMAA81]

Hassan Gomaa, D. B. H. Scott. "Prototyping as a Tool on the Specification of User Requirements," Proceedings of the Fifth International Software Engineering Conference, March 1981, pp. 333-342.

[GOMAA83]

Hassan Gomaa. "The Impact of Rapid Prototyping on Specifying User Environments," ACM Sigsoft Software Engineering Notes, Vol. 8, No. 2, April 1983, pp. 17-28.

[HAMILTON83]

M. Hamilton, S. Zeldin. "The Functional Life Cycle Model and Its Automation: USE.IT," The Journal of Systems and Software, Vol. 3, No. 1, March 1983, pp. 25-62.

Bibliography

[HARRIS85]

Harris Corporation. "Draft GKS Binding to ANSI Ada," Harris Corporation, Melbourne, FL, February 1, 1985.

[HONEYWELL84]

Honeywell Systems and Research Center. Automated Subsystems Control for Life Support Systems (ASCLSS): Applications Study Final Report Volume I, January 1984.

[HONEYWELL85]

Honeywell Systems and Research Center. "Automated Subsystems Control for Life Support Systems: Man Machine Interface Functional Design Document," April 1985.

[HOROWITZ83]

Ellis Horowitz, Alfons Kemper, Balaji Narasimhan. "An Analysis of Application Generators," Technical Report TR-83-208, University of Southern California Computer Science Department, March 1983.

[HOROWITZ84]

Ellis Horowitz, John B. Munson. "An Expansive View of Reusable Software," IEEE Transactions on Software Engineering, Vol. SE-10, No. 5, September 1984, pp. 477-487.

[IEEE83]

IEEE Computer Society. IEEE Computer Society Workshop on Software Engineering Technology Transfer, IEEE Computer Society, April 1983.

[INGALLS78]

Daniel H. H. Ingalls. "The Smalltalk-76 Programming System Design and Implementation," Proceedings ACM Fifth Annual Symposium on Principles of Programming Languages, Tucson Arizona, January 1978, pp. 9-16.

[ISSI86]

International Software Systems, Inc.. "PSDL: Prototype System Description Language," ISSI Technical Report, unnumbered, January 30, 1986.

[JONES84]

T. Capers Jones. "Reusability in Programming: A Survey of the State of the Art," IEEE Transactions on Software Engineering, Vol. SE-10 No. 5, September 1984, pp. 488-494.

[KERNIGHAN84]

Brian W. Kernighan. "The Unix System and Software Reusability," IEEE Transactions on Software Engineering, Vol. SE-10 No. 5, September 1984, pp. 513-518.

[KRUCHTEN84]

Philippe Kruchten, Edmond Schonberg, Jacob Schwartz. "Software Prototyping Using the SETL Programming Language," IEEE Software, Vol. 1, No. 4, October 1984, pp. 66-76.

[LANERGAN84]

Robert G. Lanergan, Charles A. Grasso. "Software Engineering with Reusable Designs and Code," IEEE Trans. on Software Engineering, Vol. SE-10D, No. 5, September 1984.

[LARRABEE84]

Tracy Larrabee, Chad Leland Mitchell. "Gambit: A Prototyping Approach to Video Game Design," IEEE Software, Vol. 1, No. 4, October 1984, pp. 28-38.

[LISKOV75]

Barbara H. Liskov, Stephen N. Zilles. "Specification Techniques for Data Abstractions," IEEE Transactions on Software Engineering, Vol. SE-1. No. 1, March 1975, pp. 7-19.

[MacLENNAN85]

Bruce J. MacLennan. "A Simple Software Environment Based on Objects and Relations," Naval Postgraduate School Technical Report, NPS52-85-005, April 1985.

[MANLEY83]

John H. Manley. "IEEE CS Workshop on Software Technology Transfer: Report of the Group on Technology Transfer Process and Vehicles," IEEE Computer Society Workshop on Software Engineering Technology Transfer, IEEE Computer Society, April 1983, pp. 9-10.

[MATSUMOTO84]

Yoshihiro Matsumoto. "Some Experiences in Promoting Reusable Software: Presentation in Higher Abstract Levels," IEEE Transactions on Software Engineering, Vol. SE-10 No. 5, September 1984, pp. 502-513.

[NEIGHBORS80]

James Milne Neighbors. "Software Construction Using Components," PhD Thesis, University of California at Irvine, 1980.

[NEIGHBORS84]

James M. Neighbors. "The Draco Approach to Constructing Software from Reusable Components," IEEE Transactions on Software Engineering, Vol. SE-10 No. 5, September 1984, pp. 564-574.

[ONUEGBE85a]

Emmanuel O. Onuegbe. "Functional Specifications for a Software Engineering Database Management System," Honeywell Computer Sciences Center Technical Report, CSC-85-5:8213, Honeywell Computer Sciences Center, Golden Valley, MN 55427, March 1985.

[PETERS83]

Lawrence Peters. "The Chinese Lunch Syndrome in Software Engineering Education: Causes and Remedies," IEEE Computer Society Workshop on Software Engineering Technology Transfer, IEEE Computer Society, April 1983, pp. 87-89.

[RADCS84]

RADC. "Functional Description for Rapid Prototyping System," developed for Rome Air Development Center under contract F30602-83-C-0067, 1984.

[RENTSCH82]

Tom Rentsch. "Object Oriented Programming," ACM Sigplan Notices, Vol. 17, No. 9, September 1982.

[ROSS77]

Douglas T. Ross. "Structured Analysis (SA): A Language for Communicating Ideas," IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, January 1977.

[SEN82]

ACM SIGSOFT. Software Engineering Notes: Special Issue on Rapid Prototyping, Vol. 7, No. 5, December 1982.

[SHEIL83]

Beau Sheil. "Power Tools for Programmers," Datamation, February 1983, pp. 131-144.

[SHEIL83a]

Beau Sheil. "The Artificial Intelligence Toolbox," Proceedings of the NYU Symposium on Artificial Intelligence and Business. Ed. Walter Reitman, ABLEX, 1983.

[STANDISH84]

Thomas A. Standish. "An Essay on Software Reuse," IEEE Transactions on Software Engineering, Vol. SE-10 No. 5, September 1984, pp. 494-497.

[STARS83]

Stars. Software Technology for Adaptable, Reliable Systems (STARS) Program Strategy, U. S. Department of Defense, April 1983.

[SYMBOLICS84]

Symbolics Inc.. "Lisp Flavors," Symbolics 3600 Series User's Manual, Vol. 3B, Symbolics Inc, Cambridge, MA, 1984.

[TEICHROEW77]

D. Teichroew, E. Hershey. "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information processing Systems," IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, January 1977, pp. 41-48.

[TEITELMAN84]

Warren Teitelman. "The Cedar Programming Environment:: A Midterm Report and Examination," Xerox PARC Technical Report, CSL-83-11, June 1984.

[WEBSTER77]

Merriam Co.. Webster's New Collegiate Dictionary, G. & C. Merriam Co., Springfield, MA, 1977, pp. 990.

[WEISER82]

Mark Weiser. "Scale Models and Rapid Prototyping," ACM Sigsoft Software Engineering Notes, Vol. 7, No. 5, December 1982, pp. 181-185.

[WOOD84]

William T. Wood, Elaine N. Frankowski. "CSDL: The Concurrent System Definition Language," Honeywell Computer Science Technical Report CSC-84-7:8213, April 1984.

Bibliography

[YEH84]

Raymond T. Yeh, Nicholas Roussopoulos. "Management of Reusable Software," IEEE Comcon 84, Arlington, Virginia, September 16-20, 1984, pp. 311-320.

[ZAVE79]

Pamela Zave. "A Comprehensive Approach to Requirements," COMSAC (Conf. On Software & Applications) '79, IEEE Computer Society, 1979, pp. 117-122.

[ZAVE82]

Pamela Zave. "An Operational Approach to Requirements Specification for Embedded Systems," IEEE Transactions on Software Engineering, Vol. SE-8, No. 3, May 1982, pp. 250-269.

[ZAVE84]

Pamela Zave. "An Overview of the PAISley Project-1984," unpublished technical report, AT&T Bell Laboratories, Murray Hill NJ, undated.

[ZAVE85]

Pamela Zave. "The Operational Versus The Conventional Approach to Software Development," Communications of the ACM, Vol. 27 No. 2, February 1984, pp. 104-118.

END
FILMED

5-86

DTIC